DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

0

AFIT/GCE/ENG/90D-08

MODEL-BASED REASONING IN ELECTRONIC REPAIR

THESIS

Kenneth Bruce Cohen

AFIT/GCE/ENG/90D-08

Approved for public release; distribution unlimited

AFIT/GCE/ENG/90D-08

MODEL-BASED REASONING IN ELECTRONIC REPAIR

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Kenneth Bruce Cohen, B.S.

December, 1990

Approved for public release; distribution unlimited

Preface

I would like to acknowledge everyone who had a part in helping me to complete "The AFIT experience." I appreciate the nomination I received to attend AFIT from my supervisor Ray Clodfelter, and also appreciate all the support of my co-workers at AGMC. I gratefully acknowledge the financial support from the Scientist and Engineers Civilian Career Enhancement Prog am (SECCEP) which allows civilian engineers to attend AFIT.

I cannot begin to express my appreciation to the faculty members who have shared their knowledge and experience with e. I appreciate the time and efforts of the members of my thesis committee, particularly Dr. Frank Brown, without whose encouragement this document would have never been completed.

My personal thanks are due to my friends and family. The past 18 months have put a strain on all my relationships and I am glad that everyone stuck with me. I will not try to name everyone, because I would inevitably omit someone.

My final thanks go to my new wife, Diane. We started our life's adventure together during my time at AFIT. As I told her numerous times, "If we made it through this, the rest of our lives will seem easy."

Kenneth Bruce Cohen

Table of Contents

Page

List of Figures

List of Tables

AFIT/GCE/ENG/90D-08

Abstract

Diagnosis is the process of identifying the cause for an observed behavior of a system. Model-based reasoning uses knowledge of the behavior of the component and the interconnection of the components to diagnose a system. This thesis investigates the application of model-based reasoning to the problem of isolating faulty components of an analog electronic circuit. More specifically, this thesis describes an architecture for a model- based diagnostic system for electronic modules, implements an assumption-based truth maintenance system (one of the component parts of a model-based diagnostic system), and creates the interface between the truth maintenance system and the diagnostic system.

[illegible]

MODEL-BASED REASONING IN ELECTRONIC REPAIR

I. Introduction

Diagnosis is the process of identifying the cause for an observed misbehavior of a system (4:361). Two previous AFIT thesis efforts have investigated the use of model-based reasoning to diagnose inertial guidance systems (24) (31). This thesis investigates the application of model-based reasoning to the problem of isolating faulty components of an analog electronic circuit.

These earlier theses have worked on the problem of isolating faulty modules or faulty inertial components in the Dual Miniature Inertial Navigation System (DMINS). DMINS is a U.S. Navy inertial navigation system used on the Los Angeles class fast attack submarine. This system is repaired by the Aerospace Guidance and Metrology Center (AGMC) at Newark AFB, Ohio. Once the technicians at AGMC have replaced one of the 35 modules of the DMINS unit, they troubleshoot and repair that module. The repaired module is then available as a spare for use in future system repair. Each year, AGMC repairs over 200 of these modules (29). DMINS is just one of the systems which is repaired at AGMC, the smallest of six Air Force repair depots.

Air Force Logistics Command (AFLC) has undertaken to insert artificial intelligence technology into weapons system support (17). The main thrust of this effort has been the development of diagnostic expert systems. Conventional expert systems are developed with an approach which usually requires two people to implement: an "expert diagnostician" and a "knowledge engineer." The process is described by Giarratano and Riley:

> The knowledge engineer first establishes a dialog with the human expert in order to elicit the expert's knowledge.... The knowledge engineer then codes the knowledge explicitly in the knowledge-base. The expert then evaluates the expert system and gives a critique to the knowledge engineer. This process iterates until the system's performance is judged satisfactory by the expert. (15:6)

There are three problems to this approach. First of all, the expert is usually a person with little time to devote to such a project. Second, knowledge engineering is a relatively new field, and there is a shortage of trained personnel. Third, the iterative nature of this procedure is time consuming and imprecise. Estimating the time required for an expert system project and the effectiveness of the finished system in solving diagnostic problems is difficult.

Developing expert systems for the electronic modules found in weapons systems requires that separate expert systems be developed for each electronic module. A model-based diagnostic system might provide a more efficient method for developing future diagnostic capability for weapon system modules.

Background

Traditional Diagnosis of Electronic Modules. In (3), the method currently used at AGMC to diagnose electronic modules is described:

> When a module is received in module repair, the first step is to perform a functional test. The functional test consists of simulating the electronic stim-

ulus received by the module in the inertial navigation system (INS). If the module passes functional testing, it is then returned to the INS level as a good module.

Sometimes, the functional test is sufficient to isolate the failed component. More often, the technician or engineer must further probe the module while it is on the test station in order to isolate the faulty component. If this is not sufficient, the technician or engineer may then remove power from the module, and take static resistance measurements.

Once the faulty component has been isolated, it is replaced and (the module is) tested again. This process is repeated until the module passes functional testing. (3:3)

The functional test is performed by inserting the electronic module into an interface test adapter (ITA). Each electronic module has a different ITA, which connects an automated test station (consisting of various electronic signal generators and precision measuring instruments) to the electronic module. A test program controls the test, signalling the generators to stimulate the module and the other instruments to measure the results.

Ideally, the functional test enables the technician to isolate the faulty components. This is rarely, however, true in practice. The functional test checks the correct operation of the circuit. Often little support is provided for relating functional test failures to component failures. The technicians who perform the testing often have little training on the theory of operations of the circuits and need engineering support when the limited diagnostic information available to them proves inadequate. A limited supply of engineers is available for the diagnosis of many different systems. This leads to more costly repairs since faults are not always repaired, or components which are replaced (and not reused) were not actually faulty.

Model-based Diagnosis. In his 1988 thesis, Captain Jim Skinner used a combination of an expert system and a model-based system to diagnose the DMINS unit. The result, called the Blended Diagnostic System (24:3), was the first effort to explore model-based diagnostics at AFIT. In 1989, Captain Ray Yost used a model-based reasoning tool to further develop the capability of diagnosing the DMINS system using the model-based reasoning paradigm (31).

The concept of using model-based reasoning in diagnosis can be traced back to the 1984 article *Diagnostic Reasoning Based on Structure and Behavior* by Randall Davis. In this paper, Davis "describes a system that reasons from first principles, i.e., using knowledge of structure and behavior" (4:346). In conventional diagnostic expert systems, the diagnosis is based upon a prediction of the manner in which components fail. Davis showed that a technique called *constraint propagation* could diagnose systems based only upon the knowledge of the correct behavior of the components and the knowledge of the interconnection of those components.

Johan de Kleer and Brian Williams added to this technique by diagnosing "failures due to multiple faults" and using model-based predictions "to propose measurements to localize faults" (9:97). Peter Struss and Oskar Dressler tried to integrate model-based techniques with predictions of how components might fail in (26). De Kleer and Williams also examined combining information on component failure with model-based techniques in (10). An effort to describe analog signals for representation in model-based reasoning applications was proposed by Walter Hamscher in (16). Other recent works using model-based diagnosis have been those by Jiah-shing Chen and Sargur N. Srihari in (2), Alice McKean and Anthony Wakeling in (18), and David Tong, et al., in (27) and (28).

Truth Maintenance Systems. Model-based diagnosis requires "enumerating and keeping careful track of assumptions" (4:397). The underlying machinery for keeping track of these assumptions is a *truth maintenance system.* One of the early efforts in developing a truth maintenance system was by Jon Doyle in (12). Doyle uses "the Truth Maintenance System (TMS) to determine the current set of beliefs from the current set of reasons, and to update the current set of beliefs in accord with new reasons in a (usually) incremental fashion" (12:232). Thus, the TMS does not track what is true, but what is believed based upon the current information available. The total system consists of a TMS module and a problem-solver module. The current information available is developed by the problem-solver module and the information is maintained in the TMS. Since the current beliefs are allowed to change over time, the reasoning of the TMS is called *nonmonotonic.* Johan de Kleer developed the Assumption-based Truth Maintenance System (ATMS) in 1986. The ATMS is the basis for this thesis; its theory is discussed later. Also contained in later chapters is a comparison of the TMS and the ATMS, along with a discussion of the reasons for selecting the ATMS for this thesis.

Much of the literature on model-based diagnosis describes a system which has two components. The *model-based diagnostic engine* which has information about the system being diagnosed and a *truth maintenance system* which keeps track of the consistency of the system and the inferences that have been made. This type of system requires that the diagnostic system be preloaded with equations describing the system under test. A third component might be added for electronic module repair. This third component, called the model-making module, would provide a circuit description in PSpice format (which can be generated from a graphic interface program) as input; it would yield the equations for the model-based diagnostic engine as output. These components are described further in Chapter 3.

Using the Model-Based Diagnostic System in Depot Repair. In the repair depot, a model-based diagnostic system would reside on a computer adjacent to the automatic test equipment (ATE) running the functional test. The model-making module creates a library of models for the diagnostic system to use in diagnosing the circuits which are tested on the ATE. The ATE reports to the technician whether a functional test on an electronic module is acceptable (GO) or unacceptable (NOGO). When the ATE registers a NOGO condition, the information about the test which failed is reported to the diagnostic system. The diagnostic system compiles a list of the sets of components which could have caused this observed fault. Each additional NOGO registered by the ATE would also be reported to the diagnostic system. When the functional test concludes, the diagnostic system would report any sets of components which the diagnostic system determined could have accounted for each of the functional test NOGOs. If more than one set of components could have cause the problems, then the diagnostic system would suggest additional tests to discriminate amongst the possible faults.

Research Objectives

Although some previous AFIT theses have investigated diagnosing electronic modules (30), (21), and others have investigated using model-based diagnostic techniques at subsystem level (31), (24), no previous AFIT research has investigated using model-based reasoning to diagnose electronic modules. Diagnosis of electronic modules offer to provide a large number of applications well-suited to model-based reasoning techniques. The objectives of this research were:

- to describe the architecture for a model-based diagnostic system for electronic modules,
- to generate one of the components of the model-based diagnostic system, specifically, the ATMS, and
- to create the interface between the ATMS and the diagnostic engine.

Scope

An attempt was made to use an actual (although pedagogical) analog electronic circuit. The circuit selected was an overvoltage detector from the thermoelectric signal amplifier from the DMINS (part number 67800-502-21) (11:102-103). It uses an operational amplifier to sense an overvoltage condition and provides a high output voltage if the input voltage exceeds a certain value. The circuit, shown in Figure 1, was simplified to allow for ease in understanding the equations which represent the components.

The ATMS was developed using an 80286-based personal computer. The language used was PC Scheme with the Scheme Object Oriented Programming System (SCOOPS) extension. This combination was chosen to allow the researcher to continue to work on this project after leaving AFIT. In the description of the implementation of the ATMS, de Kleer discusses an optimization of the method for computing labels (an ATMS feature discussed in a later chapter) using boolean representations of labels (6:157-158). PC Scheme provides no built-in functions for manipulating boolean numbers (such as bitwise AND

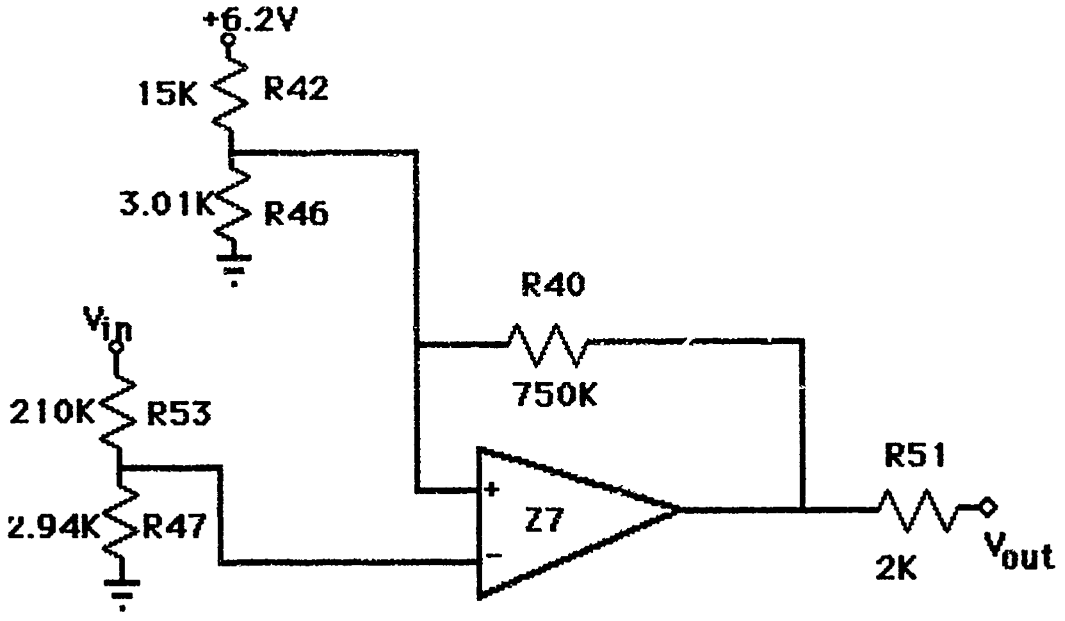

Figure 1. Overvoltage Detector

and bitwise OR), so this label computation has to be handled with symbol manipulation. This optimization could be added if this ATMS is later rehosted on a LISP machine (such as the TI Explorer). The messages which the diagnostic engine would send to the ATMS are incorporated into the system to demonstrate the operation of the ATMS.

The operation of the model-making portion of the system, namely the development of equations describing the electronic circuit, has been simulated. This portion of the system would have received a representation of the schematic as input, and provided a set of equations describing the circuit, i.e., the model of the circuit, to a model library as output.

In identifying the components of an electronic circuit which are faulty, the model-based diagnostic system only identifies the minimal sets of components which may have caused a fault. Every superset of a mimimal set of components is also a potential cause of the failure. Each possible set of components which may have caused the failure is called a *candidate.* The current method of diagnosing an electronic module would require a technician or engineer to probe the module to determine the actual fault from amongst the possible candidates identified during a functional test. No procedure was developed in this research to discriminate amongst the candidates to discover which components were actually faulty, although some methods are suggested in (10).

Assumptions

The techniques of model-based diagnosis assume that faults in the circuit are independent of each other. Catastrophic faults, which do not allow power to be supplied to the module, cannot be diagnosed by model-based techniques. Often these types of faults might be visually apparent to the diagnostician. Soldering faults, in which adjacent conductors of a component are inadvertently connected, are also not considered by these techniques.

Overview

This research investigates the application of model-based reasoning to the problem of isolating faulty components of an analog electronic circuit. The remaining chapters of this thesis discuss:

- other research on using model-based reasoning for diagnosis,
- background information on the techniques of model-based reasoning and truth maintenance systems,
- methodology used in conducting this research,
- findings of this research, and
- conclusions and recommendations of the researcher.

II. Recent Research

Introduction

Over the past few years, there has been a good deal of interest in using the AI technique of model-based reasoning on the problem diagnosing electrical and mechanical systems. Diagnostic tools based on this technique have been used to investigate the problems of diagnosing systems as diverse as digital electronic circuits, analog electrical circuits, electro-mechanical systems, and mechanical systems. Similar techniques have been applied to monitoring and correcting process control systems (13) (14). In this chapter, the various approaches to model-based diagnosis of systems are investigated.

The use of models in diagnosing systems is referred to as "deep reasoning" (24:5) or "reasoning from first principles" (22:57). Heuristic approaches (rule-based expert systems) use shallow knowledge, often based upon the experiential knowledge of an expert. Deep reasoning systems are based upon knowledge of the design and the correct behavior of a system. Model-based reasoning systems try to use a generalized diagnostic engine (9), requiring only the development of a model to troubleshoot a system.

There are two components to a model-based diagnostic tool. The first is the reasoning engine, which reasons about the model, and the second is the model of the system to be diagnosed. Research has been done on both of these components, and a discussion of them is included in this chapter.

Model-based Reasoning Engines

General Diagnostic Engine (GDE). Johan de Kleer of Xerox Palo Alto Research Center and Brian Williams of the MIT AI Laboratory have developed a model-based diagnostic engine which they call the General Diagnostic Engine (GDE) (9:97). This work is built upon the Assumption-Based Truth Maintenance System (ATMS) which de Kleer developed in 1986 (9:97).

GDE has been used in the domain of troubleshooting digital circuits. There are a number of important features of GDE:

> First, the system diagnoses failures due to multiple faults. Second, failure candidates are represented and manipulated in terms of minimal sets of violated assumptions, resulting in an efficient diagnostic procedure. Third, the diagnostic procedure is iterative, exploiting the iterative nature of diagnosis. Fourth, a clear separation is drawn between diagnosis and behavior prediction, resulting in a domain (and inference procedure) independent diagnostic procedure. Fifth, GDE combines model-based prediction with sequential diagnosis to propose measurements to localize faults. (9:97)

GDE uses the model to generate a predicted behavior. This is compared to the actual behavior of the system being diagnosed. The result is a behavioral discrepancy, which leads GDE to find possible structural discrepancies or causes of the errant behavior (9:99). A set of components whose failure could have caused the behavior observed is called a *conflict set*. Since every superset of a conflict set is also a conflict set, GDE identifies all the minimal conflict sets (9:101-107), that is, conflict sets which have no subsets that are also conflict sets. The method used by GDE to identify these minimal conflict sets is called *constraint propagation* and is discussed in Chapter 3.

Fault Isolation System (FIS). The Fault Isolation System (FIS) is a project of the Naval Center for Applied Research in Artificial Intelligence. The goal of FIS is to interface to automatic test equipment in order to generate diagnostic tests of a system. FIS has been applied to the repair of a sonar subsystem with 105 replaceable modules and to generating test trees for a radar receiver-exciter subsystem.

In order to isolate faults, FIS reasons from a model of the unit under test, information about item cost, a description of the tests available and the cost of performing each test. The order of the tests recommended is designed to provide a maximum amount of information about the fault at minimum possible cost (20). This is similar to the method used to discriminate among the minimal conflict sets as described in (5).

Types of Modeling

Knowledge representation is important to any artificial intelligence system, and model-based diagnostics are no exception. Some of the different types of models which have been developed are discussed in this section.

Structural Models. The deep reasoning portion of Captain James Skinner's Blended Diagnostic System used a model of an inertial navigation system which was based on the structure of the system under test. This model "...performs structural diagnosis through the principle of locality. The concept of locality refers to the manner in which components are connected" (24:44). This model knows only how the components are connected but not how they function.

If a component of the device under test is not functioning properly, then the inputs to that device are checked. If the inputs are functioning properly, then the device is concluded to be faulty. The knowledge of how to diagnose the device is dependent upon information about the expected outputs of the components. In this model there is no knowledge of the behavior of the components, only their interconnection to other components.

Structure and Behavior Models. A technique which has proved successful for diagnosing digital electronic circuits is *constraint suspension.* This technique uses "knowledge of structure and behavior" to diagnose a device (4:347). Captain Raymond Yost used this strategy to diagnose faults of an inertial navigation system with a tool called IDEA (31:7). To test a prospective fault, a modified model of the system is created by using a model of the entire system and excluding (suspending) the components of the prospective fault being tested. If this modified model is consistent with the observed inputs and outputs, then those components may account for the observed behavior, and they might be faulty. This technique was extended to cover multiple component failures in the General Diagnostic Engine (9). The technique of constraint suspension is discussed further in Chapter 3.

Failure Models. Constraint propagation has no inherent knowledge of the ways in which the unit under test might fail. Peter Struss and Oskar Dressler added knowledge of the manner in which the components of a system fail to their diagnostic system. Struss and Dressler argue that knowledge of how a device may fail can improve the performance of the General Diagnostic Engine (26:1318).

A system called *Sherlock*, developed by Johan De Kleer and Brian Williams, also uses *a priori* knowledge of the operating and failure modes of the components in order to pinpoint faulty components faster. This system has knowledge about the probabilities of particular failure modes and will create and test the models of more likely faults before examining the models of less likely faults (10).

Other Models. The models described above use electrical interconnection to reason about the system. There are other types of interactions between components which could be used to model the system. An unintentional solder bridge would not conform to the electrical interconnection models. A model to diagnose this type of fault would have to know which pins of the components were adjacent, or which traces on the circuit board were in proximity.

Another type of component interaction is electromagnetic. A component may create an electromagnetic field which affects the behavior of another component. Still another type of model would account for the heat generated by each component and the effect which this has on the other components. These other models are discussed in (5:339). A complete diagnostic system would necessarily contain all of these models in order to account for any possible fault.

Representating Signals

The models developed by (31) and (24) use a two-valued abstraction of signals. Signals are characterized as 'good or 'bad. Information external to the model is required to determine whether a signal is 'good or 'bad. Signals could also be abstacted to a three valued set {'high, 'good, 'low}, or additionally characterized with modifiers such as 'very-high or 'marginally-low. Signals could be classified to be within certain ranges, such as [12.2,12.5]. Digital signals could be considered by groupings of values such as 'changes (digital signal has changed value) or 'cycle (a sequence repeated itself).

This type of abstraction is described in (16). Time-varying signals could be described as 'sin(4.8khz) or 'square(400hz).

Summary

There are several approaches which have been considered in applying model-based reasoning to diagnostic systems. As in most AI problems, one of the main concerns is in knowledge representation, that is, how to represent the model and how to represent the signals flowing between the components. Different models might be used to diagnose different types of problems. Most of the research into model-based systems has used the technique of constraint suspension for the diagnostic engine. This method, which is used in this research, is discussed further in Chapter 3.

III. Theory of Operation

Introduction

This chapter describes the theory of the three components of the model-based diagostic system:

- the model-maker module,
- the diagnostic engine module, and
- the truth maintenance system module.

The model-based diagnostic system is shown in Figure 2. It is designed to diagnose an electronic circuit from a description of the schematic together with the observed inputs and outputs. The components of the system are described below.

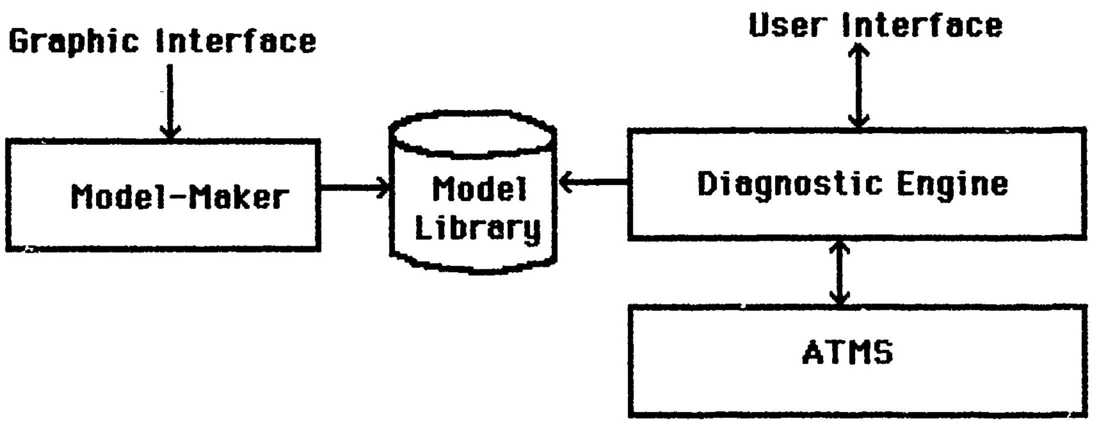

Figure 2. The Model-based Diagnostic System

The Model-Maker Module

The problem-solving systems discussed in de Kleer's ATMS (6) and Doyle's TMS (12) contains two components: the TMS (or ATMS) and the problem-solving module. De Kleer and Williams suggest a general type of problem-solving module for diagnosis using the model-based paradigm called the General Diagnostic Engine (GDE) (9:98). The GDE must have information about the behavior of the subcomponents of the system being diagnosed and how these subcomponents are interconnected. The behavior and interconnection are described in the form of mathematical equations.

The need for a mathematical description of a circuit suggests to this researcher a model-making module to automatically generate the required equations. The functions of analog and digital electronic components do not vary from one circuit to another.

The CAPFAST electronic circuit design system (19) suggests a method which may be used for the model-making module. The CAPFAST system has a graphical interface which is used to describe a circuit. The components are selected from a library of components and are graphically interconnected. In the CAPFAST system, this design is converted to a PSpice representation of the circuit. In PSpice, the components and their interconnection are defined in a netlist. An example of a PSpice-like netlist is shown in Table 1. This example describes the DMINS overvoltage circuit shown in Figure 1. The netlist consists of two parts: a partlist which maps an identifier to a component-type contained in the library, and a netlist which defines the interconnection of the circuit. The terminals of a component are based upon the type of component and are defined in the library of components.

```
* Partlist *
R53       resistor
R47       resistor
R40       resistor
R42       resistor
R51       resistor
Z7        741 op amp
R46       resistor
* End Partlist *
* Netlist *
input.0   R53.i0
gnd.0     R47.i0
gnd.0     R46.i0
6volt     R42.i0
R40.o0    Z7.i0
R53.o0    Z7.i0
R47.o0    Z7.i0
R16.o0    Z7.i0
R42.o0    Z7.i1
R46.o0    Z7.i1
Z7.o0     R51.i0
Z7.o0     R40.i0
R51.o0    output.0
* End Netlist *
```

Table 1. A PSpice Netlist

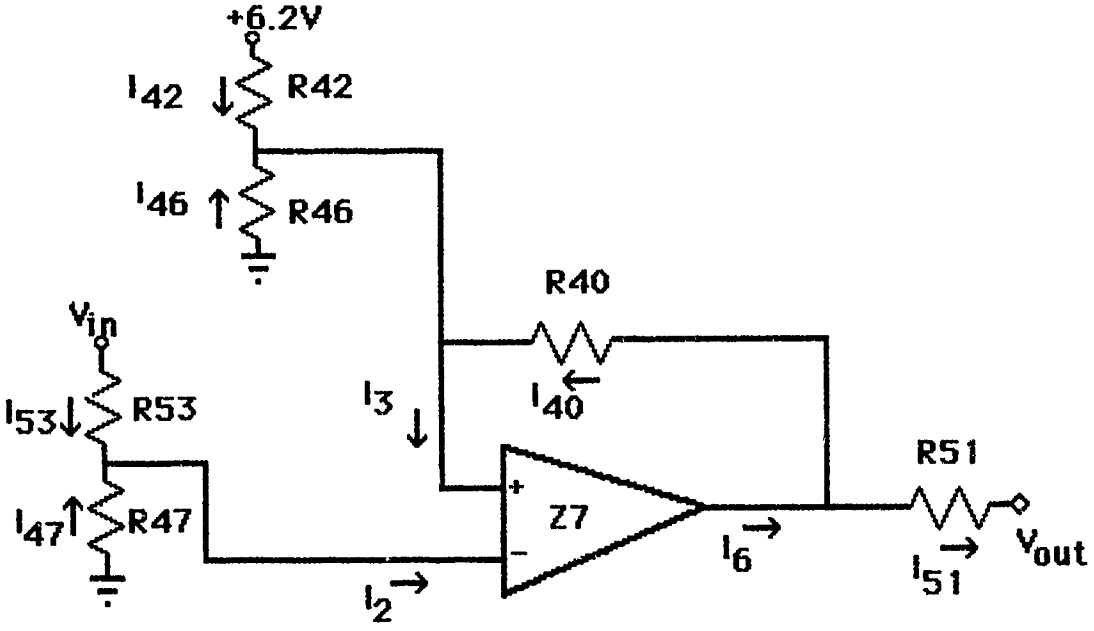

Figure 3. DMINS Overvoltage Circuit with Reference Voltages & Currents

The Spice library contains definitions of the mathematical equations relating the input(s) to the output(s). A resistor is defined to have the property that the voltage at the output is the voltage at the input minus the voltage drop, which is equivalent to the resistance value times the current value. The current through a resistor could be defined by an identifier (such as I_{47} for the current through R_{47}). Kirchoff's Current Law states that the current entering a point is equal to the current leaving a point. The model-making module would use Kirchoff's Current Law to define current restraints, which, together with the component equations, would completely define the circuit. A manually-generated definition for the circuit described in Table 1 is shown in Table 2. The equations in Table 2 would be the model of this circuit used by the diagnostic system. The last column of this table shows the component which the equation uniquely describes. Figure 3 shows the circuit with the voltage and current references as in Table 2.

1.	R_{53}	$= (V_{i_1} - V_2) \div I_{53}$	$:R_{53}$
2.	R_{47}	$= -(V_2 \div I_{47})$	$:R_{47}$
3.	R_{42}	$= (V_{i_2} - V_3) \div I_{42}$	$:R_{42}$
4.	R_{46}	$= -(V_3 \div I_{46})$	$:R_{46}$
5.	R_{40}	$= (V_6 - V_3) \div I_{40}$	$:R_{40}$
6.	R_{51}	$= (V_6 - V_K) \div I_{51}$	$:R_{51}$
7.	V_2	$= -(R_{47} \times I_{47})$	$:R_{47}$
8.	V_2	$= V_{i_1} - R_{53} \times I_{53}$	$:R_{53}$
9.	V_3	$= V_6 - R_{40} \times I_{40}$	$:R_{40}$
10.	V_3	$= -(R_{46} \times I_{46})$	$:R_{46}$
11.	V_3	$= V_{i_2} - I_{42} \times R_{42}$	$:R_{42}$
12.	V_6	$= V_7 \ if \ V_3 > V_2$	$:Z_7$
13.	V_6	$= V_4 \ if \ V_2 > V_3$	$:Z_7$
14.	V_K	$= V_6 - R_{51} \times I_{51}$	$:R_{51}$
15.	I_{53}	$= I_2 - I_{47}$	
16.	I_{53}	$= (V_{i_1} - V_2) \div R_{53}$	$:R_{53}$
17.	I_{47}	$= I_2 - I_{53}$	
18.	I_{47}	$= V_2 \div R_{47}$	$:R_{47}$
19.	I_{42}	$= (V_{i_2} - V_3) \div R_{42}$	$:R_{42}$
20.	I_{46}	$= V_3 \div R_{46}$	$:R_{46}$
21.	I_2	$= I_{47} + I_{53}$	
22.	V_7	$= -V_4 \ = \ 12$	$:PS$
23.	V_4	$= -V_7 \ = -12$	$:PS$
24.	I_3	$= I_{42} + I_{46} + I_{40}$	
25.	I_{40}	$= (V_6 - V_3) \div R_{40}$	$:R_{40}$
26.	I_{40}	$= I_6 - I_{51}$	
27.	I_6	$= I_{40} + I_{51}$	
28.	I_{51}	$= I_6 - I_{40}$	
29.	I_{51}	$= (V_6 - V_K) \div I_{51}$	
30.	V_{i_2}	$= 6.2$	$:PS$

Table 2. Mathematical Definition of Circuit

The Diagnostic Engine

Diagnosis is the process of identifying the cause for an observed misbehavior of a system (4:361). This definition divides the process of diagnosis into two steps, namely, detecting a discrepancy in the behavior of a system, and identifying the cause of the discrepancy.

In the rest of this chapter, a simple example is used to demonstrate the methodology of model-based diagnosis. The example is a circuit with three multiplier components and two adder components. The system can be described to have an output vector with two values (O_1, O_2), an input vector with six values $(I_1, \ldots, I_6)$, and the following equations relating the inputs and outputs:

$$X = I_1 \times I_2 \tag{1}$$
$$Y = I_3 \times I_4 \tag{2}$$
$$Z = I_5 \times I_6 \tag{3}$$
$$O_1 = X + Y \tag{4}$$
$$O_2 = Y + Z \,. \tag{5}$$

The outputs of the system can also be described by the equations

$$O_1 = (I_1 \times I_2) + (I_3 \times I_4) \tag{6}$$
$$O_2 = (I_3 \times I_4) + (I_5 \times I_6) \,. \tag{7}$$

The model, shown in Figure 4, is taken from (4); it is also used in other articles, such as (5) and (9).

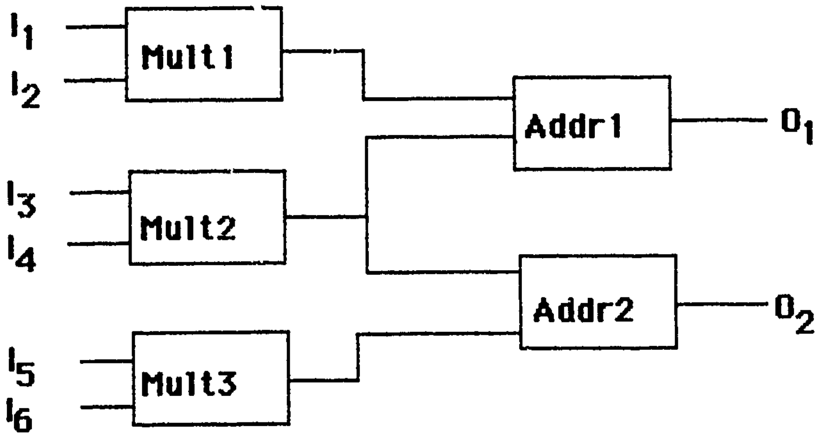

Figure 4. A Simple Example

Discrepancy Detection. In the example shown in Figure 4, the equations (6) and (7) are applied to the given inputs to obtain a predicted output. If the predicted output deviates from the observed output, then the system has misbehaved. The deviation is said to be a *symptom* of a fault. The example is shown in Figure 5 with inputs and outputs. The expected or predicted outputs are contained in parentheses ( ), and the observed outputs are contained in braces [ ]. Note that the predicted value of output O_1 is (12) and the observed value is [10].

Constraint Suspension. The next step in the diagnostic process is to identify candidates which could have caused the observed behavior, i.e., sets of components which might have failed. De Kleer defines a candidate as "a particular hypothesis for how the actual artifact (system being diagnosed) differs from the model" (9:103). For the purposes of this thesis, the following definitions will be used:

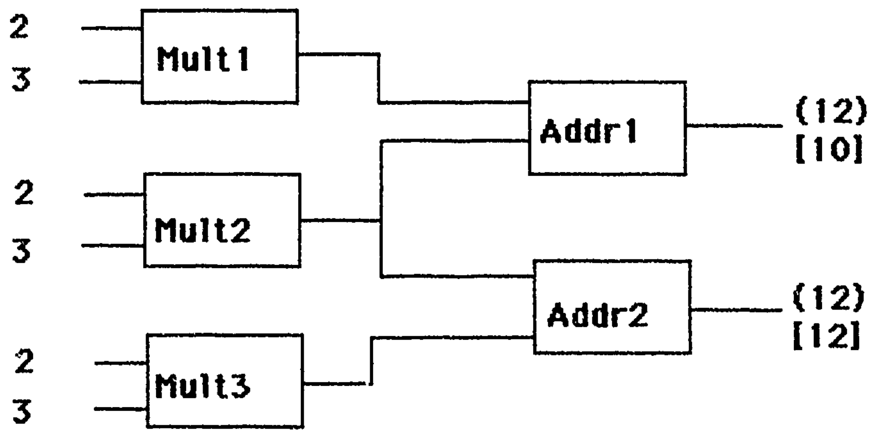

Figure 5. Example with Inputs and Outputs

- *candidate* – a set of components, the failure of which might have caused the observed behavior.
- *hypothesis* – a potential candidate, that is, a set of components being tested to see if its failure might have caused the observed behavior.

A technique known as *constraint suspension* (4:363) is used to test a hypothesis to see if it is a candidate.

Each of the components has a function, represented by a rule or equation, which forms a constraint for the system. In constraint suspension, the constraints corresponding to each of the components in a hypothesis or set of components are suspended. The hypothesis is a candidate if the system of equations is globally consistent with the observed behavior after the suspension of the equations of the hypothesized components (4:363-365).

Searching the Candidate Space. "...candidates have the property that any superset of a possible candidate for a set of symptoms must be a possible candidate as well"

(9:103). This property means that the set of every possible candidate can be described by the set of all the minimal candidates. A minimal candidate is a candidate, none of whose subsets is also a candidate. If the suspension of constraint equations for components A and B causes the system to be consistent, then the system should also be consistent if additional constraints are removed (suspended).

The method of breadth-first search can be used to identify the minimal candidates. In breadth-first search, the components are ordered into a list. A first-in first-out queue of hypotheses is constructed from each of the single components. For example, if the components are: [compl comp2 comp3], then the initial queue is [(compl) (comp2) (comp3)].

At each step of the search, the first element of the queue is tested to see if it meets the criterion of being a candidate. If the hypothesis does meet the criterion, then it is placed in the solution list. Any supersets of this hypothesis which are in the queue are then removed from the queue.

If the hypothesis is not a candidate, then the successors of that hypothesis are generated. Successors consist of all the sets which include one more component than the original hypothesis, and that this additional component is not found before any of the other components of the hypothesis in the original ordered component list. The successors are placed at the back of the queue, and are not considered until the previous elements have been considered. In this way, only the minimal candidates are generated.

When the queue is empty, the hypotheses which are in the solution list form the set of all the minimal candidates.

An Example. The example described in Figure 4 may help make this process more understandable. The components of the example system can be arbitrarily preordered in the following manner: [Mult1 Mult2 Mult3 Addr1 Addr2]. Without this preordering, the sequences (Mult1 Mult2) and (Mult2 Mult1), which represent the same set, might both be introduced into the queue. Since these sets are equivalent, only one of them must be examined. Preordering the components will prevent both of these sets from being generated. Each singleton hypothesis is then placed into the queue. The initial queue is: [(Mult1) (Mult2) (Mult3) (Addr1) (Addr2)].

The first hypothesis considered is (Mult1). Figure 6 shows the system with this hypothesis suspended. The function of Mult1 is suspended by removing equation 1 from the system. Now equation 4 will state $10 = X + 6$. The solution of this equation is $X = 4$, which is consistent with the rest of the system (once Mult1 is suspended). Therefore, (Mult1) is a candidate. No supersets of (Mult1) are in the queue and no successors of (Mult1) are generated. The queue is now: [(Mult2) (Mult3) (Addr1) (Addr2)].

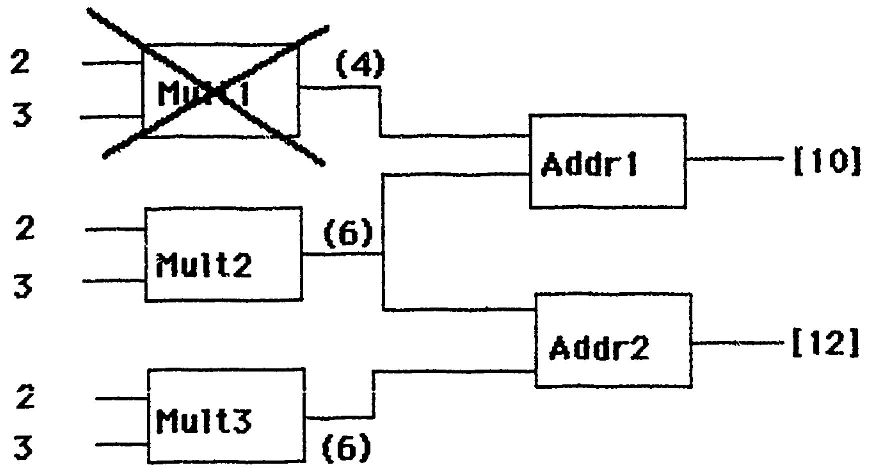

Figure 6. Mult1 Suspended

The next hypothesis considered is (Mult2). The function of Mult2 is suspended by removing equation (2) from the system. Equation (4) now reveals that $10 = 6 + Y$. This equation implies that $Y = 4$. Equation (5), however says that $12 = Y + 6$, which implies that $Y = 6$. Since the system is inconsistent (Y cannot be both 4 and 6), the hypothesis (Mult2) cannot be a candidate. The successors of the hypothesis (Mult2) are the hypotheses (Mult2 Mult3), (Mult2 Addr1), and (Mult2 Addr2). These are placed at the back of the queue. The queue then becomes: [(Mult3) (Addr1) (Addr2) (Mult2 Mult3) (Mult2 Addr1) (Mult2 Addr2)]. (Mult2 Mult1) is not a successor because of the preordering done to the component list. The search continues until the queue has been emptied. Once the queue is emptied, the solution list is: ((Mult2 Addr2) (Mult2 Mult3) (Addr1) (Mult1)).

Discriminating Amongst the Candidates. Once the search has concluded, all the minimal candidates have been identified. The final process in the procedure is to discriminate amongst the candidates to discover which component or components actually caused the fault. Although outside the scope of this research, several methods of discrimination were examined by this author, but not implemented.

The deep reasoning method employed by Skinner examines the inputs to a component which has a faulty output. If the inputs of that component are correct, then that component is faulty and must be repaired (either replaced or further broken down to subcomponents to be diagnosed). If the value of any of the inputs to a component is incorrect, then the component which provided that input is diagnosed (24:44). The input selected to be tested is based upon the cost of testing each input. This method has some difficulty in diagnosing circuits containing a loop, and can be somewhat inefficient for circuits containing many components. Also, it is not always well known what the output of specific components should be, so this information must be gathered in developing the system.

De Kleer and Williams describe a more efficient method of discrimination in (9). Whereas Skinner's method was to probe the circuit linearly, choosing the input with the least cost first, de Kleer and Williams perform a test to see which measurement reduces the entropy of the system most. The entropy is based upon the estimated number of remaining measurements needed to find the actual candidate. This is referred to as a look-ahead strategy (9:113-115). This strategy is more efficient than the linear method of Skinner, but suffers from the same problems with loops.

Assumption-Based Truth Maintenance System

The final component of the model-based diagnostic system is the Assumption-Based Truth Maintenance System (ATMS). This section will examine Jon Doyle's Truth Maintenance System (TMS), how the ATMS differs from the TMS, the theory behind the ATMS, and how the ATMS is used in the model-based diagnostic system.

Truth Maintenance System. Elaine Rich describes the Truth Maintenance System (TMS) as "...an implemented system that supports nonmonotonic reasoning (23:180)." She continues to say that the role of the TMS "...is not to generate new inferences but to maintain consistency among the statements generated by the other program (23:180)." In the model-based diagnostic system, the diagnostic engine makes inferences, and the TMS maintains consistency among the inferences which have been made.

The TMS contains nodes, each of which contains data. Each node at a given time is either believed to be true (IN), or not believed to be true (OUT). A node which is OUT may be either believed to be false, or have no justification to believe that it is true. Nodes can be moved from IN to OUT and vice versa, depending upon justifications for each node. There are two types of justifications in the TMS: support list justifications, and conditional proof justifications.

A support list justification contains an IN list and an OUT list. If each of the nodes of the IN list is IN and each of the nodes of the OUT list is OUT, then the node corresponding to the support list justification is believed (IN). If the support list justification has empty IN and OUT lists, then the node corresponding to the support list justification is always believed to be true and is called a *premise.*

The conditional proof justification has a consequent node, an IN list, and an OUT list. If the consequent node is IN, then the IN and OUT lists are checked. If the nodes of the IN list are IN and the nodes of the OUT list are OUT, then the node is believed (IN). If the consequent node is OUT, then the IN and OUT lists are not checked, and the node does not change conditions. Conditional proof justifications can be used to prevent circular arguments (23:181-183).

ATMS versus TMS. De Kleer referred to the TMS of Jon Doyle as a "conventional justification-based TMS" (6:129). De Kleer refers to his ATMS as "more-efficient than previous TMSs and has a more coherent interface between the TMS and the problem-solver without giving up exhaustivity" (6:128). In the problem of model-based diagnosis, one of the extra features of the ATMS is important.

The TMS serves three roles in this overall system. First it functions as a cache for all the inferences ever made. Thus inferences, once made, need not be repeated, and contradictions, once discovered, are avoided in the future. Second, the TMS allows the problem-solver to make nonmonotonic inferences (e.g., "Unless there is evidence to the contrary, infer A"). The presence of non-monotonic justifications requires that the TMS use a constraint satisfaction procedure (called truth maintenance) to determine what data is to be believed. Third, the TMS ensures that the database is contradiction-free. Contradictions are removed by identifying absent justification(s) whose addition to the database would remove the contradiction. (6:129)

These three roles are important to the model-based diagnostic system. The cache function allows the system to remember the inferences that have been made previously. For example, once the output of Addr1 has been determined to be 6, it is not necessary to recompute the value of the output of Addr1 for the same inputs, given the fact that Addr1 is operational. This allows inferences to be search problems, rather than computational problems. This important feature is also a part of the ATMS.

The second role of the TMS is to "allow the problem-solver to make nonmonotonic inferences" (6:129). Doyle describes "a reason for a belief ..." to consist "...of a set of other beliefs, such that if each of these basis beliefs is held, so also is the reasoned belief" (12:234). He continues that there must be either a circular argument or a "fundamental type of belief which grounds all other arguments" (12:234). The TMS therefore allows base assumptions to be made, upon which other beliefs can be reasoned. The ATMS also allows assumptions to be made.

The third role of the TMS is to maintain a contradiction-free environment. This feature is used to check consistency in the model based diagnostic system. One of the primary differences between the TMS and the ATMS is that the TMS can maintain only one contradiction-free environment at a time. A TMS *environment* (not to be confused with a

PC Scheme environment) is defined as a set of assumptions. In order to proceed to another environment, the TMS must backtrack and create different consistent environments. The ATMS keeps track of the environments where particular data are consistent. This allows the ATMS to track multiple environments simultaneously (6:129).

TMS and ATMS assumptions are usually of the form "if there is no reason to believe $\neg P$ then believe P." An assumption can therefore be retracted by giving a reason for believing $\neg P$. The TMS uses a procedure called dependency-directed backtracking to select assumptions to retract (with reason). In this procedure, the TMS "traces backwards through the reasons for conflicting beliefs ...(to find) the set of assumptions reached ...and then retracts one of the assumptions with a reason involving the other assumptions" (12:235-236).

The key difference between the ATMS and the TMS is this ability of the ATMS to deal with multiple environments. To determine the best solution, the model-based paradigm for diagnosis requires all the possible solutions to be identified and examined. As information is added to the system (derived through additional testing and probing), the previous information should be available. The multiple environments allow maintenance of all previously derived information.

ATMS. This section describes the operation of the ATMS. Some definitions may prove helpful. A node describes problem-solver datum. For example, one node might describe the datum $X = 6$. The node also contains a justification, a label, and a consequent list.

The justification describes how a node is derivable from other nodes. The problem-solver provides the justification for a node. A typical justification might be that Mult1 is operating correctly and the values of I_1 and I_2 are 2 and 3 respectively. The justification contains three components: a consequent, an antecedent, and an informant. The consequent of the justification is the node being justified, and the antecedent is a list of the nodes which justify that node. The informant provides a facility for a problem-solver description of the justification, which can be used to supply the user with a way to trace the reasoning. The informant is not used by the ATMS developed for this model-based diagnostic system.

The node also contains a label, which is a list of environments under which the node is consistent. An environment in the ATMS is a conjunction of a set of assumptions. This label contains the minimal environment, since a node is also consistent under supersets of each environment in the label.

The consequent list of a node is the list of the justifications in which the node is an antecedent. The consequent nodes must be updated when the label of a node is updated.

The ATMS maintains a list of inconsistent environments (called nogoods). The names of these environments (and the supersets of these environments) cannot be contained in the label of any node.

The ATMS has only three functions, creating a node, creating an assumption, and adding a justification to a node. Creating a node and an assumption require little work by the ATMS. The main work of the ATMS is done when a justification is added to a node. When a justification is added to a node, it is necessary to update the label of the node. The new label is "the union of all possible combinations of picking one environment from each antecedent node label" (6:151). This label is made minimal and consistent by removing any environments not minimal and also not consistent. An environment is not consistent if some subset of that environment is contained in the nogood list.

If the label of the node mentioned in the justification is changed, it is necessary to check (and update if necessary) the labels of each of the nodes in the consequent list of that node. Changing the label of a node requires that the consequents of that node be checked. When no further updates are possible, the labels are all minimal and consistent.

Using the ATMS. In the model-based diagnostic system, the diagnostic engine module communicates directly with the ATMS. The assumptions which are created are that the components of the system being diagnosed are operating correctly, and that they are not operating correctly. The nogood list contains each pair {Component X is operating correctly, Component X is not operating correctly}. In the example shown in Figure 4, the assumptions would be 'Mult1 is operating correctly', 'Mult1 is not operating correctly', 'Mult2 is operating correctly', etc. For ease in representing these assumptions, Mult1 will represent 'Mult1 is operating correctly' and ¬Mult1 will represent 'Mult1 is not operating correctly.' The nogood list would contain pairs such as {Mult1, ¬Mult1}, {Mult2, ¬Mult2}, etc.

The diagnostic engine sends information to the ATMS. Initially, assumptions are created for each component of the system. Next, premises about the input values are put into the system. A premise is a node which is always believed. It requires no justification and never receives one. Next, the diagnostic engine creates nodes for the predictive operation of the system. For example, a node is created announcing that $X = 6$ with the label

{Mult1}. In other words, if Mult1 is operating correctly, then X is 6. When the observed outputs are entered into the system, these are entered into the ATMS as premises.

If the observed outputs do not match the predicted outputs, then the diagnostic engine would begin a breadth first search of the solution space. Once a value has already been computed for a given environment (or its subset), the value is looked up in the database of the ATMS and not recomputed. The breadth first search is described above in the section entitled " Searching the Candidate Space." Once completed, the diagnostic engine would contain the minimal candidates which could have cause the anomaly. Additional information derived from further testing is directly entered into the diagnostic engine, which enters additional premises into the ATMS and repeats the process. In this way, the model-based diagnostic system determines the faulty components.

IV. Methodology

This chapter discusses a diagnostic system using the model-based reasoning paradigm which was implemented for this thesis. The portions of the system which were implemented are the assumption-based truth maintenance system (ATMS) discussed in (6) and a diagnostic engine similar to that discussed in (9).

The model-based diagnostic system and ATMS were implemented for this thesis in PC Scheme, a dialect of LISP developed by Texas Instruments. The Scheme Object Oriented Programming System (SCOOPS) was also used. A top-down design methodology was employed in developing this system.

Top-Down Design Methodology

A technique which was extensively used throughout this project is the *Bravery Principle.* This technique was discussed by Dr. Frank Brown during EENG 592 (1). The algorithm for the Bravery Principle is:

- In writing some code, include all the functionality which can be done easily.
- If some functionality requires more work, then defer this decision of how to do that work by calling a function (to be written later) to perform that work.
- After completing the code, write the code for any functions which were deferred (again using the Bravery Principle in developing this code).

The Bravery Principle is sometimes referred to as a top-down development approach (25:308). The top level functions are developed first, then the lower level functions are developed as needed. The top level functions contain broad, general instructions, and the lower level becomes increasingly detailed. This approach is good for a single developer, since the programmer can concentrate on one level of functionality at a time, deferring consideration of greater detail until a later time.

Research Methodology

Incremental Software Development. In approaching the problem of a developing a model-based diagnostic system, the author first had to identify the major component parts of the software system. The model-based system shown in Figure 2 contains three components:

- the model-making module,
- the diagnostic engine module, and
- the truth maintenance module.

These modules needed to be developed in a bottom-up manner (i.e., the truth maintenance module must be developed before the diagnostic engine module, and the diagnostic engine before the model-maker module). The model-maker module was simulated by "hard-coding" a specific example, namely the adder/multiplier problem (shown in Figure 5). A more generalized model-maker module is discussed in Chapter 3, but was not implemented during this research. The algorithms for the ATMS and the diagnostic engine are discussed in Chapter 3. The specific implementations are discussed below.

Implementation of ATMS. The ATMS which was developed for this research was specifically tailored to be used with the model-based diagnostic engine. This requires the use of the following conventions:

- Nodes contain data of the form `MULT1`, `NOT_ADDR2`, or `W_2=6` (i.e., an assumption or WIRE=value).
- An assumption will be a node of the form `MULT1` or `NOT_ADDR2` (i.e., a component is working or not working).
- An environment is a list (possibly empty) of assumptions.
- A label is a list (possibly empty) of environments.
- A justification contains a consequent (the node being justified), an informant (some description of the justification), and an antecedent (the nodes which justify the consequent). Although a slot was allowed for the informant, it was not used during this research.
- A premise is a node with no justification or label. It is assumed to be true.
- The nogood list is a list of environments which are assumed to be false.
- The consequents of a node are the nodes which are justified by that node.

In (6), three external functions are defined. These functions allow the addition of data into the node-list of the ATMS. These functions are:

- CREATE-NODE, which creates a new node.

- CREATE-ASSUMPTION, which creates an assumption as a node with the datum also contained in the label and the justification.
- ADD-JUSTIFICATION, which computes and updates the label when a node is justified.

In creating the ATMS for use with the model-based diagnostic engine, some additional functions were created to allow some specific types of entries into the node-list. Functions were also created to access the data in the node-list, reset the system, and examine all the data in the node-list. These functions are:

- CREATE-PREMISE, creating a specific type of node with no label and no justification. This is used for data which are definitely to be believed by the system (always true).
- GET-INDEX, takes the datum of a node as input and returns the whole datum if the node is an assumption, and the portion of the datum to the left of the equal sign '=' otherwise.
- GET-VALUE, takes an environment and an index and returns the portion of the datum to the left of the equal sign '=' for a node which matches the index and the environment, or returns `'nil` otherwise.
- ADD-NOGOOD, adds an environment to the nogood list.
- RESET-TMS, sets all global variables (including the node-list) to `'nil`.
- EXAMINE-ALL-NODES, returns information about each node in the node-list.

The node-list contains all the nodes (including assumptions and premises) which have been created. Each entry in the node-list contains an index and a list of nodes which correspond to this index. Therefore all the nodes which correspond to a specific index, for example I_1, can be examined in GET-VALUE to find the one which corresponds to the currently applicable environment.

One of the tasks performed by the ATMS is computing the label of a node. Each element of the label is an environment where the datum of that node is valid. The environment is a set of assumptions. The label, then, is a set of environments, each environment being a set of assumptions. A label of the current node is computed by taking the union of every combination of environments in the label of each node justifying the current node. In (6), de Kleer discusses representing assumptions by bit-vectors, thus allowing the computation of a label "by or'ing the bit-vectors. Set1 is a subset of set2 if the result of and'ing the bit-vector of set1 with the complement of set2 is zero (6:157)." PC Scheme does not have a built-in implementation of bit-vector operations, so these set operations were developed to work on symbols, not bit-vectors.

An example might help to make this clear. If the labels `((MULT1) (MULT2 MULT3))` and `((MULT1 ADDR1) (MULT2))` were combined, the result would be computed by list manipulation to be: `((MULT1 MULT1 ADDR1) (MULT1 MULT2) (MULT2 MULT3 MULT1 ADDR1) (MULT2 MULT3 MULT2))`. After removing the repetitious symbols, the label is: `((MULT1 ADDR1) (MULT1 MULT2) (MULT2 MULT3 MULT1 ADDR1) (MULT2 MULT3))`. Since the goal is to create minimal labels, any label-element (environment) which is a superset of another label-element would then be removed to yield the following label: `((MULT1 ADDR1) (MULT1 MULT2) (MULT2 MULT3))`. This label would then be compared to the nogood list to remove any label-elements which were supersets of nogood environments. In the bit-vector implementation, these computations are more efficient.

The source code for the ATMS is shown in Appendix A.

Implementation of the Model-Based Diagnostic Engine. The model-based diagnostic engine uses the method of constraint propagation discussed in (1). In constraint propagation, a module output is connected to a wire. When the output is changed, this value is propagated to the wire. The wire, in turn, propagates this value to the terminals of other modules to which this wire is connected. This causes the output of those modules to change, and so forth until no more changes occur and the system is stable. The terminal values of the system (inputs and outputs) are input as premises (since they are defined by the actual observable state of the system. The values of interior wires are inferred from rules defined by the function of the components in the model. These inferences are entered into the ATMS as nodes, justified by the known wires.

A breadth-first search function systematically suspends the function of some of the modules of the system under test. These suspended modules correspond to a hypothesis being tested. The suspended modules are unable to propagate values. If the system infers two values for a wire, then that system is inconsistent, and the corresponding hypothesis is not a candidate. If the system is consistent, then the hypothesis is a candidate and is added to the solution set.

The current environment is the conjunction of the modules not in the hypothesis with the negation of the modules in the hypothesis. For example, if the hypothesis is: `(MULT1, ADDR1)`, then the environment would be: `(NOT_MULT1 MULT2 MULT3 NOT_ADDR1`

ADDR2). To retrieve the value of one of the wires (for example, W_1), the labels of the nodes matching the wire would be searched for a node with labels consistent with this environment. A label is consistent with the environment if each element of the label is a member of the environment.

The source code for the model-based diagnostic engine is shown in Appendix B.

Implementation of the Model. Although the method of propagating values through the system is the same for any model, the method of calculating the value to propagate through a component (module) will be based upon the types of components. For this research, only simple two-input, single-output modules were considered. Each module must have a name and a flag to indicate if it is currently suspended. No calculation of values will be done if the module is suspended. The third slot in the module is an equation representing the module. The equation is a list which represents:

1. a PC Scheme function representing the module function,
2. the wire connected to input 1,
3. the wire connected to input 2,
4. the wire connected to the output, and
5. an inverse function of 1, i.e., a method of computing an input if the output and other input are known.

A PC Scheme method (called EVALUATE) is used to calculate values of the wires connected to a module (if possible), and to determine if a calculated value conflicts with the value already calculated for a wire. If the value does conflict, then that hypothesis is eliminated as a candidate. Otherwise, evaluation of the system continues.

Each module and wire is instantiated and bound to a global variable with the name of the wire. These wires and modules can then be referred to by the various functions. A list of the modules, the initial queue for the search function, and lists of the wires are also bound to global variables.

The source code for the model is shown in Appendix C.

Equipment

This research was implemented on an Epson Equity III (80286 based) microcomputer with 640 kilobytes of main memory. This equipment was chosen to allow the researcher to continue to examine this problem after completion of this thesis. The system consists of 757 lines of code, comprising 87 PC Scheme expressions.

Summary

The system which was developed for this research solves the diagnostic problem shown in Figure 5. The solution is shown in Appendix D. Appendix D also shows the node-list (displayed with the function EXAMINE-ALL-NODES) at the conclusion of the diagnostic run. Other systems can be run by changing the model. New models could be developed automatically by the model-making module, the design of which was outside the scope of this research.

V. Conclusions and Recommendations

Introduction

This research endeavored to develop the tools to study a model-based diagnostic system. The ATMS and model-based diagnostic engine were implemented in PC Scheme. This chapter examines the problems encountered during implementation, problems foreseen when the system is enlarged to solve realistic circuit problems, and suggestions for future work in this area.

Problems Encountered

In (6), de Kleer discusses representing assumptions by bit-vectors, thus allowing the computation of the union of two sets by "by or'ing the bit-vectors. Set1 is a subset of set2 if the result of and'ing the bit-vector of set1 with the complement of set2 is zero (6:157)." PC Scheme does not have a built-in implementation of bit-vector operations, so these set operations were developed to work on symbols, requiring the traversing of lists multiple times, a procedure far less efficient than bit-vector operations. The symbolic manipulation of sets was more difficult and less efficient to implement and to execute. This bit-vector implementation could be incorporated if this code is hosted on a LISP machine (such as the TI Explorer).

The majority of the effort required to keep the node-list consistent is performed in the diagnostic engine module (which corresponds to the problem-solving module of the ATMS based system. This corresponds to the result which de Kleer found in (7) and (8).

The module which communicates with the ATMS must determine which justification will be applied to the nodes calculated.

Problems Foreseen

Towards the conclusion of this research, it became impossible to load all the modules of the system along with the EDWIN editor which is part of PC Scheme. This memory shortage will make it difficult to solve larger problems.

In (31:62), Capt Yost discusses a problem in diagnosing a circuit with feedback. Many electronic circuits contain feedback loops, which present a problem to a constraint propagation method. In a feedback loop, the input of a module is dependent upon its output. In a constraint propagation system, this system may take a long time to stabilize. Additionally it is difficult to isolate the faulty component within a feedback loop.

Recommendations

Due to the high cost of repairing electronic modules and the prolonged development time for conventional expert systems, research into model-based diagnosis should be continued. The development of the problem-solver module, which can create models from a circuit description, will allow rapid creation of diagnostic systems for the myriad of electronic modules in the Department of Defense inventory. Future work needs to be done to make the ATMS more efficient (possibly by hosting the system on a LISP machine). Also the problem of feedback loops needs to be explored. The models which would be developed from an actual circuit need to be examined. The PC based version of the diagnostic system was at the limit of the physical memory of the system used. A PC based version using extended or expanded memory versions of PC Scheme should also be examined. Future AFIT thesis work should begin to approach these problems.

Conclusion

This research provides an initial development of the tools needed to conduct further research into model-based diagnosis. This system diagnoses a simple model of an imaginary circuit. The research also yielded a PC-based implementation of the ATMS. More complex circuits were not examined due to the memory limitations. The automated model-making module which develops the models from a representation of a schematic of the circuit was not attempted.

Appendix A. Code for Assumption-Based Truth Maintenance System

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                    ;;
;;  Program: An Assumption Based Truth Maintenance System (ATMS)      ;;
;;  Filename: ATMS.S                                                  ;;
;;  Author: Ken Cohen                                                 ;;
;;  Date: 10/02/90                                                    ;;
;;  Version: 2.1                                                      ;;
;;  Language: PC Scheme with SCOOPS object oriented programming       ;;
;;  System: IBM PC compatible with MS-DOS                             ;;
;;  File Processing: To be loaded after SCOOPS.FSL                    ;;
;;                   and must be used with a problem solving module   ;;
;;  Contents: Classes: node                                           ;;
;;                     justification                                  ;;
;;            Major Operations: create-node                           ;;
;;                              create-premise                        ;;
;;                              create-assumption                     ;;
;;                              add-justification                     ;;
;;                              get-value                             ;;
;;                              get-index                             ;;
;;                              reset-tms                             ;;
;;                              examine-all-nodes                     ;;
;;                              add-nogood                            ;;
;;            Global Variables: current-node                          ;;
;;                              node-list                             ;;
;;                              nogood                                ;;
;;                              current-node-list                     ;;
;;                              index                                 ;;
;;                              current-label                         ;;
;;                                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                       CLASS DEFINITIONS                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                           NODE                                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  A node is the basic unit of the ATMS.                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-class node
              (instvars datum        ;; Supplied by the problem solver
                                     ;; All datum for this project should
                                     ;;   be of the form MULT1,
                                     ;;   NOT_MULT1, or W_1=10
                                     ;;
                        label        ;; The label defines the
                                     ;;   environments where a node
                                     ;;   is believed.
                                     ;; Consists of a list of assumptions
                                     ;; Computed by the ATMS
                                     ;;
                        just         ;; Used by the ATMS to compute the
                                     ;;   label.
                                     ;; Supplied by the problem solver.
                                     ;;
                        (consq nil) ) ;; The nodes whose justifications
                                     ;;   mention this node
                                     ;;
              (options
                inittable-variables
                settable-variables
                gettable-variables ) )

(compile-class node)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                           JUSTIFICATION                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  A justification is reason that the problem solver supplies for    ;;
;;  believing the datum of a node.                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-class justification
              (instvars informant       ;; Problem solver description
                                        ;;   (not used in this effort)
                                        ;;
                        antecedent)     ;; The nodes which justify the
                                        ;;   consequent
                                        ;;
              (options
                inittable-variables
                settable-variables
                gettable-variables) )

(compile-class justification)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                            GLOBAL VARIABLES                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  CURRENT-NODE is used when looking for a node by its datum           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define current-node nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  CURRENT-NODE-LIST is used to store all the nodes corresponding to a ;;
;;  certain class (for example: all the nodes concerning the variable   ;;
;;  X are contained in the same class and would be stored into current- ;;
;;  node-list if we LOOKUP X)                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define current-node-list nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  INDEX is the class of the nodes contained in the CURRENT-NODE-LIST  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define index nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; NOGOOD contains the environments which imply false                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define nogood nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; NODE-LIST contains all the nodes created.  They are indexed by class ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define node-list nil)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; CURRENT-LABEL is used to compute updated labels                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define current-label nil)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                            MAIN PROCEDURES                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          GET-INDEX                                   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; GET-INDEX takes the datum and returns the class in which it is       ;;
;; contained.  For this thesis, datum takes the form X=1, MULT1, or     ;;
;; NOT_MULT1 (depending on whether the datum is an assumption, premise, ;;
;; or other kind of datum).  The index is the part before the "=" or    ;;
;; the entire datum.                                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (get-index datum)
  (let*
    ( (str-datum (symbol->string datum))
      (eq-posit (substring-find-next-char-in-set
                  str-datum
                  0
                  (string-length str-datum)
                  "=" )))
    (cond
      ( (null? eq-posit)
        datum )
      ( else
        (string->symbol (substring str-datum 0 eq-posit)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              CREATE-NODE                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; create-node will create a node.  The argument is the datum (which    ;;
;; must be unique).  The new node is put into the node-list either in   ;;
;; matching class, or into a new class if the class did not exist.      ;;
;; Attempts to create duplicate nodes are ignored.                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (create-node datum)
  (let
    ( (index  (get-index datum)))
    (cond
      ( (null? (lookup index))
        (set! node-list
              (cons
                (list
                  index
                  (list (make-instance node
                                       'datum datum
                                       'label nil
                                       'just  nil )))
                node-list )))
      ( else
        (cond
          ( (is-found? datum)
            (writeln "DUPLICATE NODE")
            (replace-current-node) )
          ( else
            (set! current-node
                  (make-instance node
                                 'datum datum
                                 'label nil
                                 'just  nil))
            (replace-current-node) ))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                            CREATE-PREMISE                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  create-premise creates a node which is always true.  There is no     ;;
;;  label and no justification for a premise.  There can be only one     ;;
;;  premise for any index (which can then contain no other nodes).  A    ;;
;;  premise is a special type of node which is placed in the node-list  ;;
;;  (into the correct class)                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (create-premise datum)
  (let
    ( (index (get-index datum)))
    (lookup index)
    (set! node-list
          (cons (list index
                      (list (make-instance node
                                           'datum datum
                                           'label nil
                                           'just  nil )))
                node-list ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          CREATE-ASSUMPTION                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; create-assumption is used to create assumptions.  Only the datum is   ;;
;; required as an argument.  The label and the jus[illegible]ation are also     ;;
;; equal to the datum (in proper format).  An assu[illegible]on is a special     ;;
;; type of node and is placed into the node-list (under the index)       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (create-assumption datum)
  (let
    ( (index (get-index datum)))
    (cond
      ( (null? (lookup index))
        (set! node-list
              (cons
                (list
                  index
                  (list
                    (make-instance node
                                   'datum datum
                                   'label (list (list datum))
                                   'just  (list (make-instance
                                                  justification
                                                  'informant nil
                                                  'antecedent
                                                  (list datum) )))))
                node-list )))
      ( else
        (cond
          ( (is-found? datum)
            (writeln "DUPLICATE NODE")
            (replace-current-node) )
          ( else
            (set! current-node
                  (make-instance node
                                 'datum datum
                                 'label (list
                                          (list datum) )
                                 'just  (list
                                          (make-instance
                                            justification
                                            'informant  nil
                                            'antecedent
                                            (list datum) ))))
            (replace-current-node) ))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          ADD-JUSTIFICATION                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; add-justification is used to add a justification to a node.  The    ;;
;; consequent is the node to be updated.  If the consequent is false,   ;;
;; then the label of the nodes is computed and added to the nogood      ;;
;; database.  If the consequent is a node, then the justification is    ;;
;; added to the node, the new label is computed for the node from the   ;;
;; antecedents (minimally), and the node (with the updated label) is    ;;
;; replaced in the node-list.  If the label has changed, then the       ;;
;; antecedents must have their labels updated.                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (add-justification consequent antecedent)
  (cond
    ( (eqv? consequent 'false)             ;; if consequent is 'false,
      (set! current-label nil)             ;; then compute current-label
      (compute-labels antecedent node-list) ;; and add to the nogood list
      (letrec                              ;; any environments which are
        ( (check-labels                    ;; not subsets of existing
            (lambda (lbl-lst)              ;; environments in nogood
              (cond
                ( (null? lbl-lst)
                  nil )
                ( (any-subsets? (car current-label) nogood)
                  (check-labels (cdr lbl-lst)) )
                ( else
                  (add-nogood (car current-label))
                  (check-labels (cdr lbl-lst)) )))))
        (check-labels current-label) ))
    ( (node->current-node consequent)      ;; get the node
      (send current-node set-just          ;; set the just of the node
            (cons (make-instance justification
                                 'informant  nil
                                 'antecedent antecedent)
                  (send current-node get-just)) )
      (set! current-label nil)             ;; compute the label
      (compute-labels antecedent node-list)
      (delete-supersets (send current-node get-label)
                        current-label nil) ;; get rid of the environ-
                                           ;; ments which are already
                                           ;; in the label
      (cond
        ( (null? current-label)
          (replace-current-node) )         ;; if nothing left then
                                           ;; do nothing
        ( else
          (send current-node set-label       ;; otherwise
                (append current-label        ;; add the labels which
                                             ;; were are left
                        (send current-node get-label)) )
          (replace-current-node)
          (update-nodes antecedent) ))       ;; and update the labels
                                             ;; of the nodes in the
                                             ;; antecedent
      (update-antecedents antecedent consequent) )
                                             ;; set the consq of each
                                             ;; node in the antecedent
    ( else
      (writeln "NODE " consequent " not found -- ADD-JUST") )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                             GET-VALUE                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;  GET-VALUE returns a value from the index which is consistent with    ;;
;;  the assumptions from an environment. NIL is returned if no label is ;;
;;  consistent.                                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (get-value index env)
  (lookup index)                             ;; lookup puts all the nodes
                                             ;; matching the index into
                                             ;; current-node-list

  (cond
    ( (null? current-node-list)
      nil)
    ( else
      (set! node-list (cons                  ;; put the nodes back into
                        (list index          ;; node-list
                              current-node-list)
                        node-list))
      (letrec                                ;; find gets the value
        ( (find                              ;; which matches the env
            (lambda (env nl)                 ;; from a list of nodes
              (cond
                ( (null? nl)
                  nil )
                ( else
                  (let
                    ( (labels (send (car nl) get-label)))
                    (cond
                      ( (or (null? labels)
                            (consistent-label? labels env) )
                        (send (car nl) get-datum) )
                      ( else
                        (find env (cdr nl)) ))))))))
      (find env current-node-list) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                           ADD-NOGOOD                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; add-nogood is used to add an environment to the nogood list.          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (add-nogood env)
  (set! nogood (cons env nogood)) )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          RESET-TMS                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RESET-TMS reinitializes all the global variables to nil               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (reset-tms)
  (set! current-node nil)
  (set! current-label nil)
  (set! node-list nil)
  (set! nogood nil)
  (set! current-node-list nil)
  (set! index nil) )
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                             EXAMINE-ALL-NODES                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; examine-node writes out the contents of each node in a node-list.        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (examine-all-nodes)
  (letrec
    ( (examine-nodes
        (lambda (current-nl)
          (cond
            ( (null? current-nl)
              nil )
            ( else
              (writeln "DATUM = " (send (car current-nl) get-datum)
                       "   LABEL = " (send (car current-nl) get-label) )
              (if (null? (send (car current-nl) get-just))
                  (writeln "JUST = "
                           nil
                           "   CONSQ = "
                           (send (car current-nl) get-consq) )
                  (writeln "JUST = "
                           (send (eval (car (send (car current-nl)
                                                  get-just)))
                                 get-antecedent)
                           "   CONSQ = "
                           (send (car current-nl) get-consq) ))
              (writeln "")
              (examine-nodes (cdr current-nl)) ))))
      (examine
        (lambda (nl)
          (cond
            ( (null? nl)
              nil )
            ( else
              (examine-nodes (cadar nl))
              (examine (cdr nl)) )))))
    (examine node-list) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                 SUPPORTING PROCEDURES     (PROCEDURE SUPPORTED)        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                 CONSISTENT-LABEL          (GET-VALUE)                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; CONSISTENT-LABEL returns nil if env is not consistent with one of the;;
;; labels in label-list.                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (consistent-label? label-list env)
  (letrec                         ;; a label is consistent with an
    ( (label-ok?                  ;; environment if one of the
        (lambda (lbl env)         ;; environments in the label is
          (cond                   ;; a superset of the environment
            ( (null? lbl)
              #t )
            ( (member (car lbl) env)
              (label-ok? (cdr lbl) env) )
            ( else
              nil )))))
    (cond
      ( (null? label-list)
        nil )
      ( (label-ok? (car label-list) env)
        #t )
      ( else
        (consistent-label? (cdr label-list) env) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                   UPDATE-ANTECEDENTS      (ADD-JUSTIFICATION)        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; update-antecedents will set the consq of any node mentioned as an    ;;
;; antecedent.  This procedure is used as part of the add-justification ;;
;; function                                                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (update-antecedents ant consq)
  (cond
    ( (null? ant)
      nil )
    ( (node->current-node (car ant))
      (cond
        ( (member (car ant) (send current-node get-consq))
                                        ;; if the antecedent is already
                                        ;; in the consq of a node, do
          nil )                         ;; nothing
        ( else
          (send current-node            ;; otherwise, add the antecedent
                                        ;; to the consq of the node
                set-consq (cons consq
                                (send current-node get-consq) ))))
      (replace-current-node)            ;; put the node back into the
                                        ;; node-list
      (update-antecedents (cdr ant) consq) )
    ( else
      (writeln "NODE " (car ant) " not found -- UPDATE-ANTECEDENTS")
      (update-antecedents (cdr ant) consq) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  DELETE-SUPERSETS         (ADD-JUSTIFICATION)        ;;
;;                                           (UPDATE-NODES)             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; delete-supersets will delete all the supersets from cl               ;;
;; (current-label) which are found in label (the previous label)        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (delete-supersets label cl newlbl)
  (cond
    ( (null? cl)
      (set! current-label newlbl) )
    ( (any-subsets? (car cl) label)
      (delete-supersets label (cdr cl) newlbl) )
    ( else
      (delete-supersets label (cdr cl) (cons (car cl) newlbl) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          UPDATE-NODES    (ADD-JUSTIFICATION)          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; update-nodes will compute the label for the nodes found in ants.     ;;
;; The solution will be stored in current-label.                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (update-nodes ants)
  (cond
    ( (null? ants)                    ;; if ants is empty, do nothing
      nil )
    ( (node->current-node (car ants))  ;; otherwise, get the first node
      (cond
        ( (null? (send current-node get-just))
          (replace-current-node)       ;; if there is no justification
          (update-nodes (cdr ants)) )  ;; ignore this node
        ( else
          (compute-labels              ;; compute the labels of the
                                       ;; informants of the just of
                                       ;; the node
            (send (car (send current-node get-just))
                  get-antecedent)
            node-list )
                                       ;; get rid of the repetitious
                                       ;; labels
          (delete-supersets (send current-node get-label)
                            current-label nil)
          (cond
            ( (null? current-label)    ;; if nothing is left, do
              (replace-current-node)   ;; nothing
              (update-nodes (cdr ants)) )
            ( else                     ;; otherwise, add the label
                                       ;; and add the antecedents
                                       ;; of this node the the nodes
                                       ;; to have labels updated
              (send current-node set-label
                    (append current-label
                            (send current-node get-label) ))
              (replace-current-node)
              (update-nodes (append (cdr ants)
                                    (send (car (send current-node
                                                     get-just))
                                          get-antecedent) )))))))
    ( else
      (writeln "NODE " (car ants) " not found -- UPDATE-NODES") )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                            IS-FOUND?      (CREATE-NODE)              ;;
;;                                           (CREATE-ASSUMPTION)        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; is-found? returns nil if the datum is not found in any node of the   ;;
;; node-list.  If the datum is found, then that node is removed from    ;;
;; the node-list and placed in current-node.                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (is-found? datum)
  (cond
    ( (null? current-node-list)
      nil )
    ( else
      (letrec
        ( (find
            (lambda (datum nl rest)
              (if
                (null? nl)
                nil
                (cond
                  ( (eqv? datum (send (car nl) get-datum))
                    (set! current-node (car nl))
                    (set! current-node-list (append rest (cdr nl)))
                    #t )
                  ( else
                    (find datum (cdr nl)
                          (cons (car nl) rest) )))))))
        (find datum current-node-list nil) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                   REPLACE-CURRENT-NODE     (CREATE-NODE)            ;;
;;                                            (CREATE-ASSUMPTION)      ;;
;;                                            (ADD-JUSTIFICATION)      ;;
;;                                            (UPDATE-NODES)           ;;
;;                                            (UPDATE-ANTECEDENTS)     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; replace-current-node puts the current-node-list back into the       ;;
;; node-list                                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (replace-current-node)
  (set! current-node-list
        (cons current-node current-node-list))
  (set! node-list (cons
                   (list index current-node-list)
                   node-list )) )
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                    COMPUTE-LABELS          (ADD-JUSTIFICATION)      ;;
;;                                            (UPDATE-NODES)           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; COMPUTE-LABELS adds the label of a node to the current-label.  The  ;;
;; result is the union of each environment in the current-label with   ;;
;; each environment in the label of the node (ant).  These unions are  ;;
;; minimized by removing repetitious assumptions.                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (compute-labels ant nl)
  (cond
    ( (null? nl)
      nil )
    (else
      (letrec
        ( (check-nodes
            (lambda (ant current-nl)
              (cond
                ( (null? current-nl)
                  nil )
                ( (member (send (car current-nl) get-datum) ant)
                  (update-label (send (car current-nl)
                                      get-label))
                  (check-nodes ant (cdr current-nl)) )
                ( else
                  (check-nodes ant (cdr current-nl)) )))))
        (check-nodes ant (cadar nl))
        (compute-labels ant (cdr nl)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         UPDATE-LABEL       (COMPUTE-LABELS)          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; update-label is used by compute-labels to add a label to the         ;;
;; current-label.  It does the actual union of the environments of the  ;;
;; label with the environments of the current-label.  It removes        ;;
;; repetitious environments and also removes nogood environments (with  ;;
;; remove-nogoods).                                                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (update-label label)
  (letrec
    ( (update-current-label
        (lambda (label)
          (cond
            ( (null? label)
              nil )
            ( else
              (cond
                ( (null? current-label)
                  (set! current-label label) )
                ( else
                  (set! current-label (append
                                        (mapcar (lambda (x)
                                                  (append (car label) x) )
                                                current-label)
                                        (update-current-label
                                          (cdr label)) )))))))))
    (update-current-label label)
    (set! current-label (undup current-label))
    (remove-nogoods) ))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              UNDUP         (UPDATE-LABEL)            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; undup uses remove-dupes to remove the duplicates from a list of      ;;
;; lists.                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (undup ls-of-ls)
  (cond
    ( (null? ls-of-ls)
       nil)
    ( else
      (cons (remove-dupes (car ls-of-ls))
            (undup (cdr ls-of-ls)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                    REMOVE-DUPES          (UNDUP)                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; remove-dupes will remove the items in a list which appear more       ;;
;; than once                                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (remove-dupes ls)
  (letrec
    ( (rem-dup
        (lambda (old-ls new-ls)
          (cond
            ( (null? old-ls)
              new-ls )
            ( else
              (cond
                ( (member (car old-ls) new-ls)
                  (rem-dup (cdr old-ls) new-ls) )
                ( else
                  (rem-dup (cdr old-ls) (append
                                          new-ls
                                          (list (car old-ls)) )))))))))
    (rem-dup ls nil) ))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                            SUBSET?       (ANY-SUBSETS?)              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; subset? returns true if ls1 is a subset of ls2                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define subset?
  (lambda (ls1 ls2)
    (cond
      ( (null? ls1)
        #t )
      ( else
        (if (member (car ls1) ls2)
            (subset? (cdr ls1) ls2)
            nil )))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  REMOVE-NOGOODS          (UPDATE-LABELS)                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; remove-nogoods removes the supersets of the current-label and also       ;;
;; removes the supersets of the nogood database.                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (remove-nogoods)
  (letrec
    ( (rem-supsets
        (lambda (ls new-ls)
          (cond
            ( (null? ls)
              (set! current-label new-ls) )
            ( (or
                (any-subsets? (car ls) (cdr ls))
                (any-subsets? (car ls) new-ls)
                (any-subsets? (car ls) nogood) )
              (rem-supsets (cdr ls) new-ls) )
            ( else
              (rem-supsets (cdr ls) (append
                                      new-ls
                                        (list (car ls)) `` ))))
    (rem-supsets current-label nil) ))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        ANY-SUBSETS?     (ADD-JUSTIFICATION)               ;;
;;                                         (DELETE-SUPERSETS)                ;;
;;                                         (REMOVE-NOGOODS)                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; any-subsets? returns false if the set is not a subset of a set in        ;;
;; set-of-sets and true if it is a subset of a set in set-of-sets           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (any-subsets? set set-of-sets)
  (cond
    ( (null? set-of-sets)
      nil )
    ( (subset? (car set-of-sets) set)
      #t )
    ( else
      (any-subsets? set (cdr set-of-sets)) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                     LOOKUP                 (CREATE-NODE)              ;;
;;                                            (CREATE-PREMISE)           ;;
;;                                            (CREATE-ASSUMPTION)        ;;
;;                                            (GET-VALUE)                ;;
;;                                            (NODE->CURRENT-NODE)       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; LOOKUP puts the index into index and places any nodes in the          ;;
;; node-list corresponding to the index into the current-node-list.      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (lookup indx)
  (letrec
    ( (lookup                             ;; the supporting procedure lookup
        (lambda (n-ls rest)               ;; steps through the node-list
          (cond                           ;; to find a set of nodes which
                                          ;; matches the index
            ( (null? n-ls)
              (set! current-node-list nil)
              (set! index indx)
              nil )
            ( else
              (let
                ( (pr (car n-ls)))
                (cond
                  ( (eqv? (car pr) indx)
                    (set! current-node-list (cadr pr))
                    (set! index indx)
                    (set! node-list (append (cdr n-ls) rest))
                    current-node-list )
                  ( else
                    (lookup (cdr n-ls)
                            (cons pr rest) ))))))))
    (lookup node-list nil) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;              NODE->CURRENT-NODE             (ADD-JUSTIFICATION)       ;;
;;                                             (UPDATE-NODES)            ;;
;;                                             (UPDATE-ANTECEDENTS)      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; NODE->CURRENT-NODE uses lookup to find a particular node              ;;
;; corresponding to some datum.                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (node->current-node datum)
  (let
    ( (index (get-index datum)))
    (cond
      ( (null? (lookup index))
        (writeln "INDEX " index " NOT FOUND")
        nil )
      ( (is-found? datum)
        current-node )
      ( else
        (writeln "NODE " datum " NOT FOUND")
        (set! node-list (cons (list index
                                    current-node-list)
                              node-list) )
        nil ))))
```

Appendix B. Code for Model-Based Diagnostic Engine

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; Program: Model-Based Diagnostic Engine                               ;;
;; Filename: MDE.S                                                      ;;
;; Author: Ken Cohen                                                    ;;
;; Date: 10/06/90                                                       ;;
;; Version: 2.0                                                         ;;
;; Language: PC Scheme with SCOOPS object oriented programming          ;;
;; System: IBM PC compatible with MS-DOS                                ;;
;; File Processing: Loaded after SCOOPS.FSL and ATMS.FSL                ;;
;;                  Requires a model to be loaded (such as EXAMPLE.S)   ;;
;; Running Instructions: (solve)                                        ;;
;; Description: This file contains the diagnostic engine described      ;;
;;       in de Kleer's article "Diagnosing Multiple Faults"             ;;
;;       based truth maintenance system                                 ;;
;; Contents: Classes: WIRE                                              ;;
;;           Procedures: MAKE-ASSUMPTIONS                               ;;
;;                       INPUT-PREMISES                                 ;;
;;                       EVAL-MODULES                                   ;;
;;                       OUTPUT-PREMISES                                ;;
;;                       MY-SEARCH                                      ;;
;;           Global Variables: conflict                                 ;;
;;                             solution                                 ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              SOLVE                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; SOLVE runs the system example, first creating the assumptions, then ;;
;; asserting the inputs, then calculating the expected values, then    ;;
;; asserting the outputs, then searching through the solution space.   ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (solve)
  (make-assumptions mod-lst)
  (input-premises input-wire-lst input-vector)
  (set! env mod-lst)
  (eval-modules mod-lst)
  (output-premises output-wire-lst output-vector)
  (my-search q)
  )

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          MAKE-ASSUMPTIONS                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; MAKE-ASSUMPTIONS creates the assumptions needed by the ATMS.  The   ;;
;; assumptions consist of the modules in the mod-lst and their negation ;;
;; (eg, MULT1 and NOT_MULT1).  Each module and negation pair is placed ;;
;; in the nogood list                                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (make-assumptions m-ls)
  (cond
    ( (null? m-ls)
      nil )
    ( else
      (let*
        ( (symbol (car m-ls) )
          (not_sym (string->symbol
                     (string-append "NOT_"
                                    (symbol->string (car m-ls))))) )
        (create-assumption symbol)
        (create-assumption not_sym)
        (add-justification 'false  (list symbol not_sym)) )
      (make-assumptions (cdr m-ls)) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                           INPUT-PREMISES                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; INPUT-PREMISES uses the input-wire-lst and the input-vector to      ;;
;; create-premises in the ATMS for each input.  This is analogous to   ;;
;; making an assertion into the database.                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (input-premises in-wire-lst in-vec)
  (cond
    ( (null? in-wire-lst)
      nil )
    ( else
      (create-premise (string->symbol
                        (string-append
                          (symbol->string (car in-wire-lst))
                          "="
                          (number->string (car in-vec)
                                          '(heur) ))))
      (input-premises (cdr in-wire-lst) (cdr in-vec)) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          OUTPUT-PREMISES                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; OUTPUT-PREMISES uses the output-wire-lst and the output-vector to   ;;
;; create-premises in the ATMS for each output.  This is analogous to  ;;
;; making an assertion into the database.                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (output-premises out-wire-lst out-vec)
  (cond
    ( (null? out-wire-lst)
      nil )
    ( else
      (create-premise (string->symbol
                        (string-append
                          (symbol->string (car out-wire-lst))
                          "="
                          (number->string (car out-vec) '(heur)) )))
      (output-premises (car out-wire-lst) (cdr out-vec)) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      EVAL-MODULES                                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; EVAL-MODULES causes each module to be evaluated.  The evaluation      ;;
;; causes the values to be propagated to the adjacent wires.             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (eval-modules m-ls)
  (cond
    ( (null? m-ls)
      nil )
    ( else
      (send (eval (car m-ls)) evaluate)
      (eval-modules (cdr m-ls)) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                   BREADTH-FIRST SEARCH                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                    GLOBAL VARIABLES                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define conflict nil)

(define solution nil)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                    SEARCH MODULE (my-search)                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; MY-SEARCH perform a breadth first search.  Q is initially the mod-lst;;
;; if the Q is empty, then the search is complete.  Otherwise, the      ;;
;; first element of the Q is examined to see if it is a solution (by    ;;
;; using RESET-NETWORK).  If it is a solution, then conflict will be    ;;
;; nil and the first element of Q is added to the solution list.  Any   ;;
;; supersets of this solution must be removed from the Q (my-reduce).   ;;
;; If it is not a solution, then conflict will be #t, and the children  ;;
;; of the first element of the Q are placed at the back of the Q.       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (my-search q)
  (writeln "QUEUE is: " q)
  (cond
    ( (null? q)
      (writeln "THE SOLUTION IS: " solution)
      (set! solution nil) )
    ( else
      (set! conflict nil)
      (writeln "running RESET-NETWORK with " (car q))
      (reset-network (car q))
      (cond
        ( (not conflict)
          (set! solution (cons (car q) solution))
          (writeln "NO CONFLICT - SOLUTION IS: " solution)
          (my-search (my-reduce (cdr q) (car q))) )
        ( else
          (writeln "CONFLICT - children are: " (children (car q)))
          (my-search (append (cdr q)
                             (children (car q)) )))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  MY-REDUCE                                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; MY-REDUCE will return lst with any supersets of by removed.          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (my-reduce lst by)
  (cond
    ( (null? lst)
      nil )
    ( else
      (cond
        ( (subset? by (car lst))
          (my-reduce (cdr lst) by) )
        ( else
          (cons (car lst) (my-reduce (cdr lst) by)) )))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                       CLASS & Method Definitions                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                             WIRE                                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-class wire
              (instvars name
                        (value '())
                        modules)        ;; the modules which have the
                                        ;; wire as an input
              (options
                inittable-variables
                gettable-variables
                settable-variables) )

(compile-class wire)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                   CREATE-WIRE                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define create-wire
  (lambda (name modules)
    (make-instance wire 'name name
                        'modules modules) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                   CREATE-MODULE                                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define create-module
  (lambda (name equation)
    (make-instance module 'name name
                          'equation equation) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              UPDATE                                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; UPDATE will cause evaluation of each of the modules in its argument  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define update
  (lambda (modules)
    (cond ( (null? modules)
            '() )
          (else
            (send (eval (car modules)) evaluate)
            (update (cdr modules)) ))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                   DATUM->VALUE                                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; DATUM->VALUE extracts the value from ATMS datum                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (datum->value datum)
  (cond
    ( (null? datum)
      nil )
    (else
      (let*
        ( (str (symbol->string datum))
          (len (string-length str)) )
        (string->number
          (substring str
                     (+ 1 (substring-find-next-char-in-set
                            str 0 len "=" ))
                     len )
          'e 'd )))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  DATUM->VALUE                                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; VALUE->DATUM creates a symbol in the ATMS datum format                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (value->datum index value)
  (let*
    ( (in-str (symbol->string index))
      (val-str (number->string value '(heur))) )
    (string->symbol (string-append in-str "=" val-str)) ))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        RESET-NETWORK                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; RESET-NETWORK checks a candidate.  First the input wires are set and ;;
;; the interior wires are set to nil.  Next the env is set to correspond;;
;; to the candidate.  Finally, all the modules are evaluated.           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define reset-network
  (lambda (ls)
    (set! env nil)
    (set-wires)
    (set-environment ls mod-lst)
    (eval-modules mod-lst) ))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          SET-ENVIRONMENT                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; SET-ENVIRONMENT sets env to correspond to the environment being        ;;
;; being tested.  If a module is contained in ls, then NOT_mod-name is    ;;
;; put into the environment, otherwise, mod-name is put into env.         ;;
;; ENV is used in querying the ATMS database.                             ;;
;; This function also sets or resets the suspend flag for the module.     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define (set-environment ls m-ls)
  (cond
    ( (null? m-ls)
      nil )
    ( else
      (cond
        ( (member (car m-ls) ls)
          (send (eval (car m-ls)) set-suspend #t)
          (set! env (cons (string->symbol
                            (string-append
                              "NOT_"
                              (symbol->string (car m-ls)) ))
                          env )))
        ( else
          (send (eval (car m-ls)) set-suspend nil)
          (set! env (cons (car m-ls) env)) ))
      (set-environment ls (cdr m-ls)) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              SET-WIRES                                 ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; SET-WIRES will put the input values on the input wires, the output     ;;
;; values on the output-wires, and nil on all interior wires.             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define set-wires
  (lambda ()
    (letrec
      ( (set-inputs
          (lambda (inp-wires inp-vec)
            (cond
              ( (null? inp-wires)
                nil )
              ( else
                (send (eval (car inp-wires))
                      set-value
                      (eval (car inp-vec)))
                (set-inputs (cdr inp-wires) (cdr inp-vec)) ))))
        (set-interior
          (lambda (int-wires)
            (cond
              ( (null? int-wires)
                nil )
              ( else
                (send (eval (car int-wires)) set-value nil)
                (set-interior (cdr int-wires)) ))))
        (set-outputs
          (lambda (out-wires out-vec)
            (cond
              ( (null? out-wires)
                nil )
              ( else
                (send (eval (car out-wires))
                      set-value
                      (eval (car out-vec)))
                (set-outputs (cdr out-wires) (cdr out-vec)) )))))
      (set-inputs input-wire-lst input-vector)
      (set-interior interior-wire-lst)
      (set-outputs output-wire-lst output-vector) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              CHILDREN                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; CHILDREN returns all the children of a list.  The children are the ls;;
;; concatenated with each of the singular modules which follow it in the;;
;; mod-lst.                                                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define children
  (lambda (ls)
    (let*
      ( (last (last ls))
        (rest (rest last mod-lst)) )
      (combine ls rest) )))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  MISC UTILITY FUNCTIONS                              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              LAST                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; LAST returns the last element of a list.                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define last
  (lambda (ls)
    (if (null? (cdr ls))
        (car ls)
        (last (cdr ls)) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                              REST                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;   ;;;
;; REST returns all the elements of a list after a certain element LAST ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define rest
  (lambda (last ls)
    (if (eq? last (car ls))
        (cdr ls)
        (rest last (cdr ls)) )))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                             COMBINE                                  ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; COMBINE will put a list back together.                               ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define combine
  (lambda (ls rest)
    (if (null? rest)
        '()
        (cons (append ls (list (car rest)))
              (combine ls (cdr rest)) ))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                           CxR DEFINITIONS                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (first ls)
  (car ls) )

(define (second ls)
  (cadr ls) )

(define (third ls)
  (caddr ls) )

(define (fourth ls)
  (cadddr ls) )

(define (fifth ls)
  (car (cddddr ls)) )
```

Appendix C. Code for Adder/Multiplier Model

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                                                      ;;
;; Program: Model-Based Diagnostic Engine Example                       ;;
;; Filename: EXAMPLE.S                                                  ;;
;; Author: Ken Cohen                                                    ;;
;; Date: 10/06/90                                                       ;;
;; Version: 1.0                                                         ;;
;; Language: PC Scheme with SCOOPS object oriented programming          ;;
;; System: IBM PC compatible with MS-DOS                                ;;
;; File Processing: Loaded after SCOOPS.FSL, ATMS.FSL, and MDE.FSL      ;;
;; Description: This file contains the example of the adder/multiplier  ;;
;;       problem found in de Kleer's article "Diagnosing Multiple       ;;
;;       Faults" .                                                      ;;
;; Contents: Procedure:  EVALUATE - (method for module)                 ;;
;;           Global Variables: MULT1     I_1       W_1                  ;;
;;                             MULT2     I_2       W_2                  ;;
;;                             MULT3     I_3       W_3                  ;;
;;                             ADDR1     I_4       O_1                  ;;
;;                             ADDR2     I_5       O_2                  ;;
;;                                       I_6                            ;;
;;                             mod-lst                                  ;;
;;                             q                                        ;;
;;                             input-wire-lst                           ;;
;;                             input-vector                             ;;
;;                             interior-wire-lst                        ;;
;;                             output-wire-lst                          ;;
;;                             output-vector                            ;;
;;                             env                                      ;;
;;                                                                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                   EVALUATE (method belonging to module)              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; (send module-name EVALUATE) will propagate the values of through     ;;
;; a module.  If two terminals of the module are known, and the third   ;;
;; is not known, then the third wire is calculated and communicated to  ;;
;; the other modules connected to this wire.  If the third wire is      ;;
;; known, then the value in the ATMS database (for the environment) is  ;;
;; compared to the value computed.  If the values are the same, then    ;;
;; continue, otherwise this is a conflict and set! conflict (which will ;;
;; cause this test to fail).                                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define-method (module evaluate)
               ()
               (let* ((fcn   (first  equation))
                      (in1   (second equation))
                      (in2   (third  equation))
                      (out   (fourth equation))
                      (fcn-1 (fifth  equation))
                      (in1-dat (get-value in1 env))
                      (in1-val (datum->value in1-dat))
                      (in2-dat (get-value in2 env))
                      (in2-val (datum->value in2-dat))
                      (out-dat (get-value out env))
                      (out-val (datum->value out-dat)) )
                 (if (not suspend)
                     (cond ( (and in1-dat in2-dat)
                             (if out-dat
                                 (cond
                                   ( (eqv? out-val
                                           ((eval fcn) in1-val in2-val))
                                     nil )
                                   ( else
                                     (set! conflict #t)
                                     (add-nogood env) ))
                                 (let*
                                   ( (val ((eval fcn) in1-val in2-val))
                                     (dat (value->datum out val)) )
                                   (create-node dat)
                                   (add-justification
                                     dat
                                     (list
                                       name in1-dat in2-dat ))
                                   (update (send (eval out)
                                                 get-modules)) )))
( (and in1-dat out-dat)
      (let*
        ( (val ((eval fcn-1) out-val in1-val))
          (dat (value->datum in2 val)) )
        (create-node dat)
        (add-justification
          dat
          (list
            name out-dat in1-dat ))
        (update (send (eval in2)
                      get-modules)) ))
( (and in2-dat out-dat)
      (let*
        ( (val ((eval fcn-1) out-val in2-val))
          (dat (value->datum in1 val)) )
        (create-node dat)
        (add-justification
          dat
          (list
            name in2-dat out-dat ))
        (update (send (eval in1)
                      get-modules)) ))))))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         COMPONENT and WIRE Definitions                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define MULT1 (create-module 'MULT1 '(* I_1 I_2 W_1 /)))
(define MULT2 (create-module 'MULT2 '(* I_3 I_4 W_2 /)))
(define MULT3 (create-module 'MULT3 '(* I_5 I_6 W_3 /)))
(define ADDR1 (create-module 'ADDR1 '(+ W_1 W_2 O_1 -)))
(define ADDR2 (create-module 'ADDR2 '(+ W_2 W_3 O_2 -)))

(define I_1 (create-wire 'I_1 '(MULT1)))
(define I_2 (create-wire 'I_2 '(MULT1)))
(define I_3 (create-wire 'I_3 '(MULT2)))
(define I_4 (create-wire 'I_4 '(MULT2)))
(define I_5 (create-wire 'I_5 '(MULT3)))
(define I_6 (create-wire 'I_6 '(MULT3)))
(define W_1 (create-wire 'W_1 '(ADDR1)))
(define W_2 (create-wire 'W_2 '(ADDR1 ADDR2)))
(define W_3 (create-wire 'W_3 '(ADDR2)))
(define O_1 (create-wire 'O_1 '()))
(define O_2 (create-wire 'O_2 '()))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      SOME GLOBAL VARIABLES                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; These global variables define the model being diagnosed              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(define mod-lst '(MULT1 MULT2 MULT3 ADDR1 ADDR2))

(define q '((MULT1) (MULT2) (MULT3) (ADDR1) (ADDR2)))

(define input-wire-lst '(I_1 I_2 I_3 I_4 I_5 I_6))

(define input-vector '(2 3 2 3 2 3))

(define interior-wire-lst '(W_1 W_2 W_3))

(define output-wire-lst '(O_1 O_2))

(define output-vector '(10 12))

(define env nil)
```

Appendix D. Results of the Diagnosis

```
QUEUE is: ((MULT1) (MULT2) (MULT3) (ADDR1) (ADDR2))
running RESET-NETWORK with (MULT1)
NO CONFLICT - SOLUTION IS: ((MULT1))

QUEUE is: ((MULT2) (MULT3) (ADDR1) (ADDR2))
running RESET-NETWORK with (MULT2)
CONFLICT - children are: ((MULT2 MULT3) (MULT2 ADDR1) (MULT2
ADDR2))

QUEUE is: ((MULT3) (ADDR1) (ADDR2) (MULT2 MULT3) (MULT2 ADDR1)
           (MULT2 ADDR2))
running RESET-NETWORK with (MULT3)
CONFLICT - children are: ((MULT3 ADDR1) (MULT3 ADDR2))

QUEUE is: ((ADDR1) (ADDR2) (MULT2 MULT3) (MULT2 ADDR1) (MULT2 ADDR2)
           (MULT3 ADDR1) (MULT3 ADDR2))
running RESET-NETWORK with (ADDR1)
NO CONFLICT - SOLUTION IS: ((ADDR1) (MULT1))

QUEUE is: ((ADDR2) (MULT2 MULT3) (MULT2 ADDR2) (MULT3 ADDR2))
running RESET-NETWORK with (ADDR2)
CONFLICT - children are: ()

QUEUE is: ((MULT2 MULT3) (MULT2 ADDR2) (MULT3 ADDR2))
running RESET-NETWORK with (MULT2 MULT3)
NO CONFLICT - SOLUTION IS: ((MULT2 MULT3) (ADDR1) (MULT1))

QUEUE is: ((MULT2 ADDR2) (MULT3 ADDR2))
running RESET-NETWORK with (MULT2 ADDR2)
NO CONFLICT - SOLUTION IS: ((MULT2 ADDR2) (MULT2 MULT3) (ADDR1)
                            (MULT1))

QUEUE is: ((MULT3 ADDR2))
running RESET-NETWORK with (MULT3 ADDR2)
CONFLICT - children are: ()

QUEUE is: ()

THE SOLUTION IS: ((MULT2 ADDR2) (MULT2 MULT3) (ADDR1) (MULT1))
```

```
NODE-LIST

DATUM = O_2=12   LABEL = ()
JUST = ()   CONSQ = (W_3=8)

DATUM = I_3=2   LABEL = ()
JUST = ()   CONSQ = (W_2=6)

DATUM = MULT1   LABEL = ((MULT1))
JUST = (MULT1)   CONSQ = (W_1=6)

DATUM = ADDR2   LABEL = ((ADDR2))
JUST = (ADDR2)   CONSQ = (W_3=8 O_2=12)

DATUM = MULT3   LABEL = ((MULT3))
JUST = (MULT3)   CONSQ = (W_3=6)

DATUM = I_2=3   LABEL = ()
JUST = ()   CONSQ = (W_1=6)

DATUM = ADDR1   LABEL = ((ADDR1))
JUST = (ADDR1)   CONSQ = (W_2=4 W_1=4 O_1=12)

DATUM = NOT_ADDR2   LABEL = ((NOT_ADDR2))
JUST = (NOT_ADDR2)   CONSQ = ()

DATUM = W_2=4   LABEL = ((MULT1 ADDR1))
JUST = (ADDR1 O_1=10 W_1=6)   CONSQ = (W_3=8)

DATUM = W_2=6   LABEL = ((MULT2))
JUST = (MULT2 I_3=2 I_4=3)   CONSQ = (W_1=4 O_2=12 O_1=12)

DATUM = NOT_MULT2   LABEL = ((NOT_MULT2))
JUST = (NOT_MULT2)   CONSQ = ()

DATUM = I_5=2   LABEL = ()
JUST = ()   CONSQ = (W_3=6)

DATUM = MULT2   LABEL = ((MULT2))
JUST = (MULT2)   CONSQ = (W_2=6)

DATUM = I_1=2   LABEL = ()
JUST = ()   CONSQ = (W_1=6)

DATUM = NOT_MULT1   LABEL = ((NOT_MULT1))
```

```
JUST = (NOT_MULT1)   CONSQ = ()

DATUM = I_4=3   LABEL = ()
JUST = ()   CONSQ = (W_2=6)

DATUM = I_6=3   LABEL = ()
JUST = ()   CONSQ = (W_3=6)

DATUM = NOT_ADDR1   LABEL = ((NOT_ADDR1))
JUST = (NOT_ADDR1)   CONSQ = ()

DATUM = W_1=6   LABEL = ((MULT1))
JUST = (MULT1 I_1=2 I_2=3)   CONSQ = (W_2=4 O_1=12)

DATUM = W_1=4   LABEL = ((ADDR1 MULT2))
JUST = (ADDR1 W_2=6 O_1=10)   CONSQ = ()

DATUM = O_1=10   LABEL = ()
JUST = ()   CONSQ = (W_2=4 W_1=4)

DATUM = NOT_MULT3   LABEL = ((NOT_MULT3))
JUST = (NOT_MULT3)   CONSQ = ()

DATUM = W_3=8   LABEL = ((MULT1 ADDR1 ADDR2))
JUST = (ADDR2 O_2=12 W_2=4)   CONSQ = ()

DATUM = W_3=6   LABEL = ((MULT3))
JUST = (MULT3 I_5=2 I_6=3)   CONSQ = (O_2=12)
```

Bibliography

1. Brown, Frank M. Class Lecture in EENG 592, Introduction to AI, Fall Quarter 1990. Air Force Institute of Technology (AU), Wright-Patterson AFB OH.

2. Chen, Jiah-shing and Sargur N. Srihari. "Candidate Ordering and Elimination in Model-based Fault Diagnosis." In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1363–1368, August 1989.

3. Cohen, Kenneth B. "Application of Knowledge Based Systems in the Maintenance of Inertial Navigation System Electronics." In *Proceedings of Thirteenth Biennial Inertial Navigation Systems Conference*, October 1987.

4. Davis, Randall. "Diagnostic Reasoning Based on Structure and Behavior," *Artificial Intelligence*, *24*(1-3):347–410 (December 1984).

5. Davis, Randall and Walter Hamscher. "Model-based Reasoning: Troubleshooting." In *Exploring Artificial Intellligence: Survey Talks from the National Conference on Artificial Intelligence*, San Mateo, California: Morgan Kaufman Publishers, Inc., 1988.

6. de Kleer, Johan. "An Assumption-based TMS," *Artificial Intelligence*, *28*(1):127–162 (1986).

7. de Kleer, Johan. "Extending the ATMS," *Artificial Intelligence*, *28*(1):163–196 (1986).

8. de Kleer, Johan. "Problem Solving with the ATMS," *Artificial Intelligence*, *28*(1):197–224 (1986).

9. de Kleer, Johan and Brian Williams. "Diagnosing Multiple Faults," *Artificial Intelligence*, *32*(1):97–130 (1987).

10. de Kleer, Johan and Brian Williams. "Diagnosis with Behavioral Modes." In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1324–1330, August 1989.

11. Department of the Navy. *Depot Level Electronic Detail Technical Manual for Inertial Navigation System AN/WSN-1(V)2*, August 1974. NAVSHIPS 0967-529-7010.

12. Doyle, Jon. "A Truth Maintenance System," *Artificial Intelligence*, *12*(3):231–272 (1979).

13. Dvorak, Daniel and Benjamin Kuipers. "Model-Based Monitoring of Dynamic Systems." In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1238–1243, August 1989.

14. Gallanti, Massimo *et al.* "A Diagnostic Algorithm based on Models at Different Level of Abstraction." In *Procee[illegible]gs of the Eleventh International Joint Conference on Artificial Intelligence*, page[illegible] 50–1355, August 1989.

15. Giarratano, Joseph and Gary Riley. *Expert Systems - Principles and Programming.* Boston: PWS-KENT Publishing Company, 1989.

16. Hamscher, Walter. "Temporally Coarse Representation of Behavior for Model-based Troubleshooting of Digital Circuits." In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 887–393, August 1989.

17. HQ AFLC AI Working Group, "AFLC Artificial Intelligence Objectives." Letter, January 1987. Wright-Patterson AFB OH.

18. McKean, Alice and Anthony Wakeling. "Fault Diagnosis in Analogue Circuits Using AI Techniques." In *Proceedings of the 1989 IEEE International Test Conference*, pages 118–123, 1989.

19. Phase Three Logic, Inc. *CapFast Electronic Circuit Design CAE User's Guide*. Release 2.1.

20. Pipitone, Frank. "The FIS Electronics Troubleshooting System," *COMPUTER, 19* (1986).

21. Ramsey, 1Lt James E. *Diagnosis: Using Automatic Test Equipment and an Intelligent Expert System*. MS thesis, AFIT/GE/ENG/84D-53, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1984 (AD-A151-918).

22. Reiter, Raymond. "A Theory of Diagnosis from First Principles," *Artificial Intelligence*, *32*(1):57–95 (1987).

23. Rich, Elaine. *Artificial Intelligence*. New York: McGraw-Hill Book Company, 1983.

24. Skinner, Capt James M. *A Diagnostic System Blending Deep and Shallow Reasoning*. MS thesis, AFIT/GCE/ENG/88D-5, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1988 (AD-A202 547).

25. Sommerville, Ian. *Software Engineering*. Wokingham, England: Addison-Wesley Publishing Company, 1989.

26. Struss, Peter and Oskar Dressler. "'Physical Negation' – Integrating Fault Models into the General Diagnostic Engine." In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1318–1323, August 1989.

27. Tong, David *et al.* "Diagnosing an Analog Feedback System Using Model-Based Reasoning." In *Proceedings of the Annual AI Systems in Government Conference*, pages 290–295, March 1989.

28. Tong, David *et al.* "Diagnostic Tree Design with Model-Based Reasoning." In *Proceedings of the 1989 IEEE Autotestcon*, pages 161–167, September 1989.

29. Vaught, Timothy. Personal Interview, May 1990. DMINS System Engineer.

30. Wunz, Capt Donald R. *Diagnosis of Analog Electronic Circuits: A Functional Approach*. MS thesis, AFIT/GCS/ENG/86M-3, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1986 (AD-A172 718).

31. Yost, Capt Raymond E. *Application of Model Based Reasoning to Diagnosis of Faults in Inertial Navigation Equipment*. MS thesis, AFIT/GCS/ENG/89D, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1989 (AD-A215 561).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE

Model-Based Reasoning in electronic repair

5. FUNDING NUMBERS

6. AUTHOR(S)

Kenneth B Cohen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology,
WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/90D-08

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Approved for public release: distribution unlimited

12a. DISTRIBUTION/AVAILABILITY STATEMENT

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Diagnosis is the process of identifying the cause for an observed behavior of a system. Model-based reasoning uses knowledge of the behavior of the component and the interconnection cf the components to diagnose a system. This thesis investigates the application of model-based reasoning to the problem of isolating faulty components of an analog electronic circuit. More specifically, this thesis describes an architecture for a model-based diagnostic system for electronic modules, implements an assumption-based truth maintenance system (one of the component parts of a model-based diagnostic system), and creates the interface between the truth maintenance system and the diagnostic system.

14. SUBJECT TERMS

Model-Based reasoning electronic repair, electronic diagnosis truth maintenance, model-based diagnosis

15. NUMBER OF PAGES 107

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL